Design and Initial Implementation of a Distributed XML Database

Francesco Pagnamenta

A dissertation submitted to the University of Dublin,

in partial fulfilment of the requirements for the degree of

Master of Science in Computer Science

September 2005

Declaration

I declare that the work described in this dissertation is, except where otherwise stated, entirely my own work and has not been submitted as an exercise for a degree at this or any other university.

Francesco Pagnamenta

Dated: September 9, 2005

Permission to Lend and/or Copy

I agree that Trinity College Library may lend or copy this dissertation upon request.

Francesco Pagnamenta

Dated: September 9, 2005

Acknowledgements

I would like to thank my supervisor Declan O'Sullivan for his guidance throughout this project. Thanks also to Ruaidhri Power of the Knowledge and Data Engineering Group (KDEG) for his help.

Special thanks to my family and friends, and to the Balzli family for their support here in Dublin.

Finally, I would like to thank the NDS class for an enjoyable year.

Francesco Pagnamenta
University of Dublin
September 2005

Abstract

Techniques for distributed relational database systems have been well researched and developed. This is unsurprising given that the relational database model itself has been in development since the early 1970s. In recent years, XML has emerged as an excellent simple format for structuring and exchanging data, and the problem of using relational databases or specially designed "native" XML databases for managing XML data has also been gradually addressed. However, although some research has gone into providing techniques for different parts of the distributed XML database problem, very little research has gone into trying to put these techniques together in order to create a distributed XML database platform.

This project investigates the design issues that need to be addressed for the implementation of such a distributed XML database platform. It surveys the state-of-the-art of XML-based technologies that can be adopted and mechanisms that can be used.

The dissertation proposes a layered architecture with a data access infrastructure at the bottom layer able to integrate different types of databases supporting XML. Acting on the top of the access infrastructure, three main functional components are proposed: a distributed transaction manager, a distributed query processor, and a distributed schema manager. Additionally, support for distributed transactions has been designed and implemented.

The project includes an initial implementation of the system. The evaluation of the implementation shows that an XML-based multidatabase system is conceivable. However, it emerged that some techniques required for achieving an efficient XML distributed database have to be enhanced.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

The eXtensible Markup Language (XML) [1] is a data model language for documents which was created to structure, store and send information. It was originally designed to deal with large-scale electronic publishing, but it has become a popular text format for data exchange across the Web and other networks.

Considering that it became a WWW Consortium (W3C) [2] recommendation in 1998, a surprising number of software vendors have adopted this standard and its success appears to continuously grow. In fact, it has been widely adopted for a wide range of applications such as health-care, manufacturing, financial services, government and publishing sectors. XML and XML based standards such as Web Services seems to emerge as the de-facto mechanism for exchanging structured information between organizations and more generally applications.

Because of its success, there is the increasing need to store XML data as it is, in order to skip the data conversion process required to use traditional databases. Nowadays, many popular XML-enabled databases[1] provide the ability of storing and retrieving XML data through a data format converter. Alternatively, some database products and open-source initiatives are designed to accommodate data in a native

[1]Database Management Systems that do not use a hierarchical data model but they are extended with some sort of data model converter (e.g. Relational DB with an XML connector)

format (native XML databases[2]). Although the research in this area is in the early stages, XML native database systems are making their appearance into the world of academia and the IT market.

Motivation

Oracle, one of the major vendors of databases, claims on its website that "in the fast moving world of IT technology, the W3Cs XML standards rank second in terms of popularity behind ANSI/ISOs SQL standard". Nowadays, relational databases are considered the most deployed repository systems, but it is not excluded that in the future XML-native databases will be more popular. It is true that the research in this area is picking up producing a number of studies investigating various aspect of this data format and related technologies, from the lowest levels (e.g. concurrent access to XML documents) to the higher ones (e.g. involving Web Services [3]).

The emergence of technologies designed to operate in a distributed environment and relying on W3Cs recommendations has pushed the research community to study the distribution of XML data. However, because basic concepts are not fully available, not much research has been carried out considering a distributed environment.

The motivation behind this project is linked to the current state of this novel research field in which many mechanisms have not been studied in detail and/or commonly recognised. In particular, apart to some extent for technologies oriented toward Web Services, it has not been found any relevant work describing the issues that have to be addressed to distribute XML repositories. Those research topics include distributed transaction processing, query processing and global schema on XML-oriented multi-database systems.

[2]Apart from being a marketing term, it has never had a formal technical definition [20]. A native XML database can be considered a database designed following a hierarchical data model and typically accessible with XML-based data manipulation languages.

Goals

This project aims to investigate several aspects related to the distribution of XML documents on data sources. Particular emphasis is put on distributed transaction processing. Related aspects such as query processing and global schema representations are also considered.

The project is application oriented since it aims to design an XML-based distributed database system. The first phase of the project produces a state-of-the-art of the research area surveying technologies that are involved. Then, a platform is designed developing the concept illustrated in Figure 1.1. The platform features three main components: a distributed transaction manager, a distributed query processor, and global schema manager. Those components reside on top of a software infrastructure providing location transparency to the remote databases on which data is physically stored. In the example, the platform integrates three database management systems of which two of them uses a different data models other than XML (relational and object), but they both hold a data model converter.
Provided the global schema, a client application can therefore query/modify the platform's data as if it was a local database. Global data consistency is guaranteed by distributed transactions, while location transparency is provided by mechanisms implemented in the distributed query processor.

Additionally, the project's objective include a partial implementation of the platform permitting a more concrete investigation of the studied methods as well as an evaluation of the adopted techniques.

Contributions

The main contributions of this work can be summarised as follows:

- Design of an XML-based multi-database system describing how a transaction manager, a query processor, and a schema manager can cooperate to provide the services required by external applications.

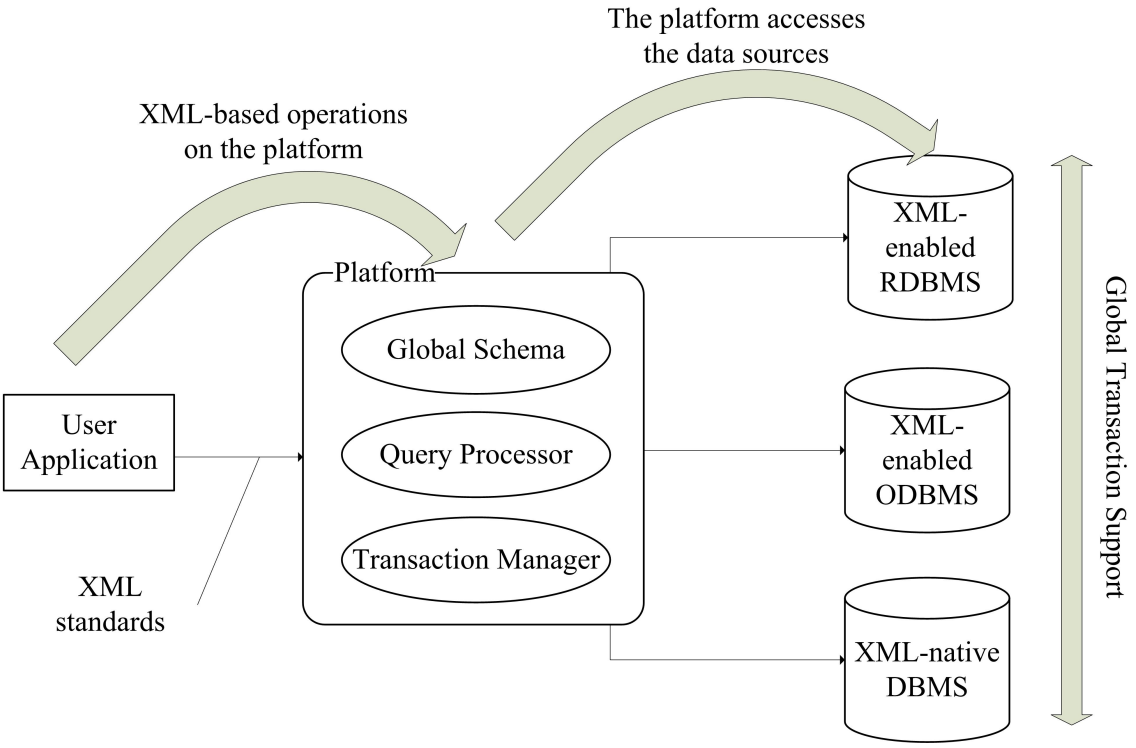

Figure 1.1: Vision of the system.

- A partial implementation of the platform featuring a transaction processor and a basic query processor. The platform accesses two database products using different access methods.

- The implementation of a simple connectivity supporting distributed transactions. Hence, a database management system (not supporting distributed transactions) is integrated into the platform using the developed client/server communication system as well as a distributed transaction module operating on the top of the repository.

- An evaluation of the implemented platform indicating that an XML multi-database system is conceivable.

Dissertation Outline

Chapter 2 introduces the research field and presents the recent advances that have been achieved in recent years. It also briefly reports an overview of techniques used for previous databases that can be re-used in an XML environment. A number of technologies and systems are also presented.

The design of the platform is reported in Chapter 3. It is initially presented the main architecture of the system with its major components. Every component is than described in the subsequent sections.

Chapter 4 explains what of the platform designed in Chapter 3 has been actually implemented. Chapter 5 contains two experiments for evaluating the implemented system. Chapter 6 lists future work in this area and draw conclusions.

Chapter 2

Background and State-of-the-Art

This chapter reports some fundamentals of XML database systems as well as the state-of-the-art techniques related to this new technology. As already mentioned earlier in this report, the research on XML databases is in the early stages. Only recently the scientific community has started to investigate this field, usually taking advantage of previous well-studied database technologies to use as a base to advance novel mechanisms applicable to this area.

2.1 The eXtensible Markup Language (XML) on Databases

XML appeared in the late nineties following several similar formats having same principles and purposes. Although it became rapidly popular among the scientific and commercial communities, XML is not a revolutionary idea; it was even not a new idea at that time. It came along with the boom of the Web and its applications. XML was introduced as a standard at the right time by an independent body, the WWW Consortium (W3C). Its predominant employment as a data exchange format on the Internet and elsewhere pushed the development of XML data repositories.

XML data definition languages

Document Type Definition (DTD) [4] defines a document structure with a list of legal elements. Although its simplicity, it lacks of important features required by the increasing number of applications that rely on the XML standards. For instance, DTD does not include a direct way to define types (e.g. integer, string, etc.). In order to overcome this and other limitations, in 2001 the W3C approved a new definition language named XML Schema [5]. XML Schemas provide a powerful mean for defining the structure, content and semantics of documents. The syntax, unlike DTDs, is expressed in XML allowing, theoretically, XML Schemas to be processed as their instances.

Structured, Semistructured, and Unstructured Data

The information stored in a database is known as structured data because it has to obey strict definition rules. Typically, when defining tables on a relational database, for each column, it has to be rigorously defined the type and, if required, its constraints. However, some applications may require a more relaxed approach where new data items can be added at any time. This type of data is described as semistructured (sometimes also referred as self-describing data). A key difference between structured and semistructured data is how schema constraints are defined and applied: structured data complies with schema directives whereas in semistructured data the information that is normally associated with a schema is contained within the data. Finally, a third category, known as unstructured data, has very limited indications of the type of data (e.g. an HTML page).

From a database perspective, one of the key decisions to be made is either to use a structured or an unstructured storage. XML-enabled and XML-native databases often provide the ability to store both categories. For a company, it might be necessary to simply store an entire document, no matter what its structure is. In contrast, applications might need to validate data against a schema before storing it. On Tamino XML Server [6] documents can be stored in collections residing on

databases. Both documents and collections are defined through XML Schema annotations. Whenever inserting or modifying data, the operation can be performed only if it is in accord with the schema constraints.

The semantics expressed by schemas can also be used for optimisation purposes. Semantic query optimisation relies on constraints defined on the database schema to modify a query into another query that is more efficient to execute (query rewriting). [24] explains how to achieve a higher degree of concurrency when performing transactions by exploiting the semantics expressed in DTDs.

XML Processing

A software module called an XML processor is used to read XML documents and provide access capabilities to their content and structure. An XML processor is typically operating on behalf of another software module which provides services to external entities. The specification of an XML processor describes its behaviour in terms of how it reads XML data and which kind of information it provides to the application. The two dominant specifications for handling XML documents are the Document Object Model (DOM) [7] and the Simple API for XML (SAX) [8]. DOM, which is a W3C standard, provides an object tree-based representation of the document; whereas SAX, a de facto standard, provides an application interface that exploits events occurring on a XML data stream model (e.g. open tag, closed tag, end of the document, etc.). The two processors are chosen according to the application requirements. SAX is more indicated to pass through a document in read-only mode. In contrast, DOM features read and update functionalities but it is known to be quite slow because of the system resources it needs. Since both specifications are designed to be platform- and language-independent, they are included in most programming languages.

The Extensible Stylesheet Language Family (XSL) [9] is a set of recommendations for defining XML document transformations and presentations. It embodies a language for transforming XML (XSL Transformations, XSLT), an expression language used by XSLT to access or refer to parts of an XML document (the XML Path

Language, XPath [10]), and a language for formatting information for paginated presentations (XSL Formatting Objects, XSL-FO). As for XML Schemas, XSL is expressed through an XML syntax.

A weakness of XML processing involves its performance impact on systems, principally because its operations are processing intensive. Parsing is therefore considered one of the major barriers for the development of high-performance XML-based technologies including XML databases [26] in which, as for any database, high-speed data access is a crucial concern.

XML data manipulation

In [30] a group of researchers in the field reports the experience in designing and implementing XML query languages. It emerged that two communities are contributing to the development of such language: the database community which is more concerned about the requirements of large repositories, and the document community which put more emphasis on integration and full-text search on single documents. This scenario may suggest the difficulties the W3C have to face to eventually come up with a solution that optimally satisfy both communities.

XL [11] is a language produced by the document community. It is quite similar to the W3C's XPath. The XML-QL [12] language introduced by the W3C was another solution providing data manipulation capabilities.

More recently, the W3C XQuery [13] standard integrates the features of previous languages and currently, looking at the features of XML-native or -enabled databases, it seems to be a good solution (at least for querying data). XQuery relies on the XPath language to access part of an XML document.

An XQuery feature to mention is the ability of construct query results by materialising XML data with a high level of flexibility. This is achieved with the FLWOR expression principle (pronounced 'flower' standing for For, Let, Where, Order by, Return) that supports iteration and binding of variables to intermediate results.

The W3C has defined the XQuery Update Facility Requirements in order to make XQuery a full data manipulation language, just like SQL for relational databases. But the fact remains that, at this stage, there is not a commonly agreed language

or language extension to update XML documents. Quite a lot of works have been carried out either to extend existing access languages or to propose new ones.

The XML:DB Initiative [14] aims to develop technology specifications for managing data on XML Databases. In the XML:DB framework, the XUpdate Project is intended to specify an Update Language for XML Documents via an XML syntax, the goal of the XML Database API Project is to develop an unique programming interface for XML Databases, and, finally, a working group has designed a common syntax and semantic for performing tasks on XML repositories (the Simple XML Manipulation Language).
Few open-source databases have adopted the technologies proposed by the XML:DB (e.g. eXist [15] and Apache Xindice [16]) initiative but it appears the commercial database vendors tend to use a proprietary language while waiting for W3C standards. Oracle XML DB [17] includes an SQL-like extension for access and update XML documents. The recent release 10 also fully supports XQuery. A DBMS may also provide other kind data manipulation such as DOM and XSL. Oracle XML DB provides a DOM access interface to manipulate XML data stored on the database.

2.2 XML Databases

A database is a collection of shared data. A Database Management System (DBMS) is a software that allows databases to be defined (e.g. data types, structures, constraints, etc.), constructed (i.e. populating the database), and manipulated (i.e. querying and updating data) [47]. In few words, a DBMS provides all tools necessary for managing a database.
A data model is the level of abstraction that hides low-level mechanisms controlling the native data representation over the physical storage. Data definition and manipulation is then performed via high-level operation applied on the data model. A very common conceptual data model is the relational one which organises data in tables and relations among items contained in tables. A data definition language (DDL) is expressed via a syntax suitable to represent the data model. A data ma-

nipulation language (DML) consists of two operation sets; data querying and data updating. In relational databases the structured query language (SQL) is used to both query and update a database. As already mentioned previously, up to now, the hierarchical data model does not dispose of a standard data manipulation language that integrates querying and updating facilities.

2.2.1 Moving Toward XML Databases

Recent advancements in the development of XML native databases management systems [20] such as Tamino [31] or Timber [32] confirms the trend: XML documents will not only be exchanged, but they will also be stored. In addition to the entrance of such databases in the world market, most of the commercial XML-enabled databases include converters. Because of this, users may be unlikely to abandon technologies that have been studied for a long time in the past and, as it is the case of the relational database technology, they are based on strong theoretical basis.

The difference between XML-native and XML-enabled may be quite vague, especially because it is believed that the term native XML databases was introduced for marketing reasons [20]. In any case, XML-enabled databases can be considered repositories that were not initially designed to store data in a hierarchical manner, but they provide an additional data format converter in order to support the XML standard and related technologies. [20] reports a possible definition of a native XML database based on the following three concepts:

- it defines a (logical) model for an XML document, and stores and retrieves documents according to that model.
- it has an XML document as its fundamental unit of (logical) storage, just as a relational database has a row in a table as its fundamental unit of (logical) storage,
- and it is not required to have any particular underlying physical storage model.

Several vendors of native XML databases claim their product support transactions (and presumably support rollbacks). However, locking is often at the level of entire documents, rather than at the level of individual nodes, so multi-user concurrency can be relatively low.

The research community is carrying out a noticeable effort to investigate efficient and reliable ways to publishing relational data as XML and vice-versa [33] [34]. Despite the achievements, in some cases, mapping the model is not enough. For example, running a transaction virtually following a hierarchical data model but actually on a relational database having locking mechanisms designed for relational databases might cause locks to be applied with an unnecessary granularity, leading to performance degradation. This example is known in the literature as pseudo-conflict that results in low inter-transaction parallelism [23].

Surveys have shown that data represented in XML and stored in a text file is three times the size of data in Java objects or in relational tables [40]. Adding to this the fact that the use of XML generates higher overhead comparing to other representations, XML itself is not the best data representation for databases. Hence, database developers have to devise techniques that make the XML data representation more adapted for this purpose. Other than reducing the disk space required, various optimisations should be applied including memory management, XML parsing optimisations, node searching optimisations, and type conversion [40].
To improve data access, xml databases support the creation of indices on stored data. Indices, just as other DBMS, can be used to improve the speed of query execution. The details of what can be indexed and how indices are applied will vary widely between products, but most support the feature in some form. Another feature required for some applications is the ability of DBMSs reacting when some events occur (active databases). A number of DBMS includes some form of trigger in their products; so does Tamino XML server.

2.2.2 Transaction Processing on XML Data

Transactions are a key concept to guarantee reliability of information systems and data consistency in the presence of system failures and interleaved access to shared data. They must be carried out so that their effect on data is serially equivalent. In order to support transactions, a DBMS has to implement a series of mechanisms that may depend on the storage model adopted on the database.

Storage and locking mechanisms

At present, there are essentially three possibilities of storing XML documents [24]. The simplest option and, from the performance point of view, also the worst, is to use an ordinary file system. In this scenario, locks are applied to entire documents causing poor concurrent access. The second alternative is to rely on an existing object or relational database. In the case of relational database systems, there are different translation mechanisms. Even if the translation technique is well engineered, it is not always the same for concurrent access since different document may share tables resulting in a too restrictive locking mechanism. The only translation scheme, in which tables are not shared, stores XML documents in Character Large OBjects (CLOBS). But again, locking is applied at the document level.
The third option is to adopt or implement an XML base management system (XBMS). In brief, it is possible to re-use well-studied techniques but, while they still guarantee serializability, they generally do not allow a sufficient degree of concurrency.
A native XML native DBMS attempts to implement node-level locking. Node-level locking is hard to achieve since it usually requires locking a parents node, which in turn requires locking its parent, and so on back to the root, ending up locking the entire document.

So, studied in depth in the past, lock mechanisms returned in the centre of attention in the context of XML. Grabs and al. [23] describe DGLOCK, a lock protocol acting at the application level, and the respective transaction manager XMLTM.

XMLTM, which was build on the top of a relational database, allows running transactions at low ANSI isolation degrees and to release database locks early, preserving the same semantics of the original transaction. In [24], it is presented an evaluation of four different core protocols for synchronising access and modifications of XML document collections. Generally speaking, there are quite a lot of lock mechanisms that have been proposed in the literature, having often in common the adoption of XPath as a means to define where the lock has to be applied (path locks). In the work reported in [25], Dekeyser and Hidders improve upon earlier work by introducing a new conflict scheduler for XML databases that uses path lock conflict rules.

2.2.3 XML Distributed Databases

XML data exchange among applications means decentralisation of information and as a consequence the distribution aspect of XML data is destined to play an important role. Application such as Web services, e-commerce applications, or the management of large-scale directories are nowadays deployed on the Internet. In addition, the need of interoperability among distributed applications makes XML a good candidate acting as a universal language that every system can comprehend. Nevertheless, distributing data over several sites implies many challenges [43] including heterogeneity, openness, security, scalability, failure handling, concurrency, and transparency. Those concerns have to be addressed like it has been done for other system in the past. The research community is investigating those concerns piece by piece completing the puzzle that will make XML databases reliable as its predecessors.

From an application point of view, the principle of a distributed database or multidatabase remains unchanged. In an ideal scenario, the conceptual architecture looks like Figure 2.1: the application client disposes of a global schema view provided by an integration component (middleware) which is connected to one or more DBMSs.

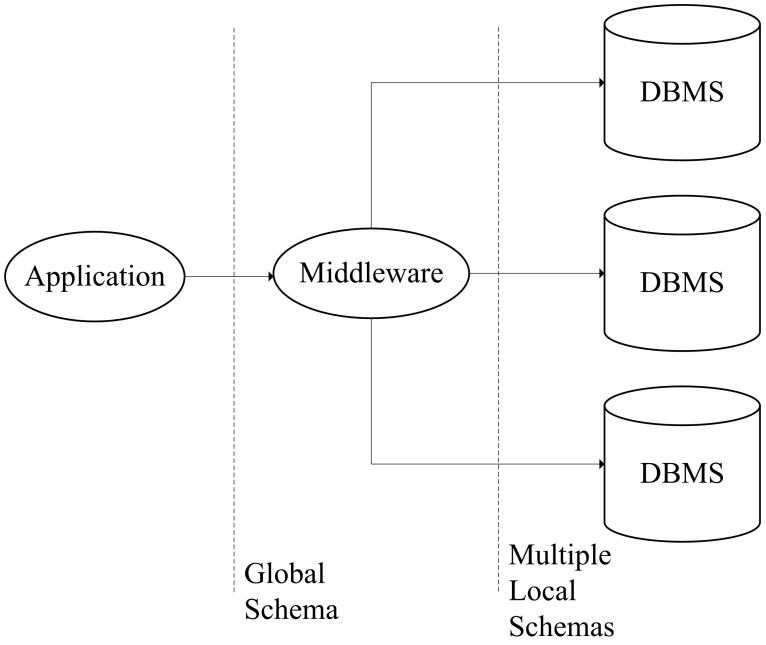

Figure 2.1: Global schema view

The DBMSs can be of different vendors and different data model, each of them disposing of a local schema. The global schema may include XML Schemas that are not the same as local ones. But a global XML Schema can be computed from local ones. In other words, the global schema is a view of local schemas. The middleware is charged for managing the connections to remote databases providing a so called location transparency [46]. Thus, the middleware provides all the conversion routines to integrate the connected systems.

Concerning transactions, although the design theory of distributed databases may seem complex, it is actually surprisingly simple [46]. The concurrency control at each site is either a strict two-phase commit locking pessimistic or an optimistic control (provided by DBMSs). The synchronization at commit time among different sites is then managed by a two-phase commit protocol. Global deadlocks, which might arise when adopting a pessimistic control, can be detected using timestamps, waits-for-graphs, or timeouts.

Previous work

Although the distribution of XML repositories is undoubtedly important, so far, only few papers have been published in this field. A possible explanation is that researchers are concentrating on more fundamental mechanisms for local systems, to be eventually extended in a distributed framework. Only recently, the research community has started to focus on this field mainly because of the popularity of distributed XML applications such as Web Services. There are also efforts for putting forward new technologies to facilitate the distribution of XML documents such as XML Fragment, a W3C recommendation enabling to exchange portions of documents.

The work presented in [22] proposes a complete framework designed for distributed and replicated dynamic XML data. In [21] the researchers focus on storage efficient index structures suitable for query evaluation in a distributed context; additionally, they describe the principles of a distributed query processor based on the concept of index shipping. Again, several works has been proposed to integrate XML and relational databases in a distributed fashion (usually following relational design patterns) but not in a pure XML context.

Distributed schema

The simplest way to distribute XML data is to organize them in documents and collections residing at different sites. The organization of data may depend on various factors such as transaction processing requirements or cost models. Consequently, the data modelization process may help to make some decisions regarding how to fragmentate data and how it will be distributed (e.g. horizontal/vertical/mixed fragmentation for the relational model). During the background research, it was not found a data modelization guideline for a hierarchical data model.

Distributed Query Processing

The metrics for evaluating distributed query processing are usually the execution cost and the response time [44]. An inefficient query optimizer could have serious

repercussions affecting the overall system performance. The aim of a query optimizer is to minimize redundant and unnecessary operations and find an optimal execution plan in order to perform the query as fast as possible. The relational model can rely on well-studied methodologies aiming to transform a query in a set of more efficient sub-operations executed in a computed order (e.g. relational normalization theory). [36] proposes an algebra for XML query optimization. [35] describes the query processor and its query optimizer having query execution strategies based on logical and physical query plans, database statistics, and a cost model. They also describe how to use heuristics for optimization purposes.

2.3 Distributed Transaction Processing

A distributed transaction involves a set of software entities which rely on an atomic commit protocol in order to reach a common agreement regarding the final outcome. A distributed transaction can be flat (i.e. one to many) or nested (i.e. one to many, many to many). Typically, a distributed transaction has a coordinator (or manager) and one or more participants.

2.3.1 Implementing Distributed Transactions

Implementing distributed transaction means ensuring the following [46]:

- **Atomic Termination**. Either all participants of a distributed transaction must commit or all must abort.
- **No Global Deadlocks**. There must be no global (distributed) deadlocks involving multiple sites.
- **Global Serialization**. There must be a (global) serialization of all transaction (distributed and local)

The followings sections cover these three points.

Atomic termination

To achieve global atomicity, a distributed transaction can commit only if all of its sub-transactions agree to commit. Because participants are located on different sites, an atomic commit protocol has to be adopted in order to reach a global consensus concerning the final outcome of the global transaction. The measures which are part of an atomic commit protocol are set according to a model defining unexpected events of different entity that can occur at any time. Failures models have been extensively researched in the past.

The **Two-Phase Commit Protocol** [29] was proposed in 1978 to face the strict requirements of distributed transactions; if one part of the transaction has failed, the overall transaction must be aborted.
The protocol is composed of two phases. In the first phase, a voting session takes place in order to assess whether the whole transaction can be committed or has to be aborted. In the second phase, every participant in the transaction carries out the decision agreed in phase one. Once a participant has voted, it can not change its mind and it must eventually, even in case of failure, carry out its part of the transaction according to the global decision. A participant is said to be in a *prepared* stated if it is ready to commit its part of the transaction.
The 2PC is a blocking protocol since, in case of site failures on the software entities involved in a distributed transaction, resources may be blocked until the overall system is repaired. A non-blocking alternative called three-phase commit protocol (3PC) has been put forward. It can handle site failures by introducing a third phase permitting participants to find out the final decision of the transaction from other participants that may have received this or other useful information from the coordinator, before being unavailable.
The two-phase commit protocol ties together software modules that may be provided by different vendors. For databases, a transaction manager (coordinator), which can operate in a middleware, access a resource manager provided by, for instance, a DBMS. The X/Open standard [19] has been conceived, among other things,

to encourage interoperability defining a set of function and messages formats needed in a transactional scenario.

A Peer-to-Peer Atomic Commit Protocol might also be adopted to achieve the atomic commitment of transactions. A set of messages are used by nodes to coordinate a distributed transaction throughout two phases. A node associated with an application program starts a transaction by defining a so called sync-point manager. By exchanging messages, the initiator creates sync-nodes on other nodes causing sync-points to spread over the network and eventually carry out the global transaction.

Distributed deadlocks

Distributed deadlock can arise in a distributed environment involving transactions. They have to be either prevented or detected and resolved. The simplest way to detect deadlocks is by using timeouts. However, it is difficult to choose an appropriate timeout interval. Most deadlock detection schemes operate by finding cycles in graphs representing the transaction (wait-for graph).

Global serialization

The simplest way to carry out a distributed transaction is to run transactions one after the other in a serial order. Unfortunately, this serial execution results in an unacceptably small transaction throughput. One way to improve performance over serial execution and yet achieve isolation is to make sure that schedulers are serializables. In a distributed environment, not only sub-transactions have to be serializable, but there must be also some sort of global serialization over all participants. But, surprisingly, global serializability can be achieved with no extra-mechanisms. In fact, if each participant uses a strict two-phase locking protocol (implemented in almost every DBMS supporting local transactions) to serialize transaction locally, and global transactions are committed using a two-phase commit protocol, then global transactions are (globally) serializable in the order in which they are committed [46].

2.3.2 X/Open Distributed Transaction Processing (DTP) Model

The X/Open Distributed Transaction Processing model [19] is a software architecture that allows multiple application programs to share resources provided by multiple resource managers, and allows their work to be coordinated into global transactions. The X/Open DTP model comprises five basic functional components:

- an Application Program (AP), which defines transaction boundaries and specifies actions that constitute a transaction
- Resource Managers (RMs) such as databases or file access systems, which provide access to resources
- a Transaction Manager (TM), which assigns identifiers to transactions, monitors their progress, and takes responsibility for transaction completion and for coordinating failure recovery
- Communication Resource Managers (CRMs), which control communication between distributed applications within or across TM domains
- a communication protocol, which provides the underlying communication services used by distributed applications and supported by CRMs.

X/Open DTP publications based on this model specify portable APIs and system-level interfaces which facilitate interoperability. Figure 2.2 capture the interactions among AP, RMs, and TM.

The Application Program (AP) is the software component that specifies a sequence of operations involving database resources. It generally represents the client application. The Transaction Manager is charged to control transactions by addressing the concerns presented previously, either globally or locally. A resource Manager (RM) may represent a DBMS and all its additional features.

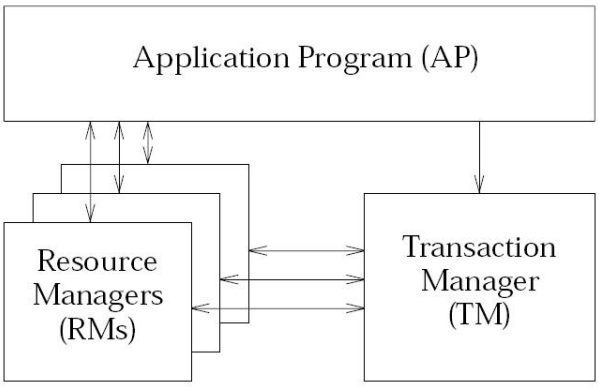

Figure 2.2: DTP model - source: opengroup.org

2.3.3 Examples of Distributed Transaction Technologies

This section briefly describes two examples of distributed transaction technologies that can be used to implement a distributed transaction system. The first is a more traditional approach while the latter operate on the top of the more recent technology of Web Services.

Java Transaction API (JTA)

JTA is part of the J2EE (Java 2 Enterprise Edition) framework. It describes transactional interfaces that provide application developers with a service model conceived to concentrate on the business logic of the application and have the transactional logic delegated to the underlying software infrastructure. The model includes a Java mapping of the industry standard XA interface based on the X/Open CAE Specification.

An advantage of the JTA is that the four ACID properties are always maintained, whereas, it can be argued that this is not an open model. In fact, the developer might encounter interoperability issues when propagating a set of transactions to other transaction services that do not share the same model.

WS-Coordination and WS-Transaction

Since Web Services rely on a stateless communication model, traditional transactional frameworks relying on remote method invocation are not applicable. This has been a major barrier to the deployment of application that uses Web Services.
Web Services Transaction (WS-Transaction) [41] is a new technology introduced to provide transactional capabilities to the Web Services framework. WS-Transaction is a specification which extends the Web Services Coordination (WS-Coordination) specification to define cooperation among systems in order to support atomic transactions. WS-Coordination is a coordination framework to enable distributed participants to agree on a global outcome depending on their individual activities. WS-Transaction defines coordination types, such as Atomic Transaction, which use the WS-Coordination framework to define rules which both the coordinator and participants must adhere to during their communications.
The main advantage of using this framework [42] is linked to the open nature of Web Services. WS-Coordination capitalises on the portability of the Simple Object Access Protocol (SOAP), used to call WS-Coordination operations. Therefore the atomic transaction service of this framework (WS-AtomicTransaction) is language and platform independent.

Chapter 3

Design

Based on the concepts and technologies presented in chapter 2, this chapter analyses the requirement of a distributed XML database putting forward a conceivable architecture. The proposed architecture is than examined in terms of its functional components which are subsequently described.

3.1 Vision

The idea is to design and implement and integrated XML-based distributed database supporting a global schema. The platform is designed to offer a high degree of transparency allowing application clients to access data in an uniform way. DBMSs fulfilling minimal requirements can therefore be integrated within the system by implementing interfaces. The client access mechanisms are conceived to be language- and system-independent by exploiting existing or upcoming XML-family standards. Figure 3.1 illustrate the overall concept of the system.

Client applications query the platform knowing the global schema. The platform process the user transactions containing queries defined in a standard access language and it performs a series of queries and sub-transactions to distant databases to retrieve/update data according to the client query. The application client may perform a series of queries executed in a distributed transaction so that global atomicity is guaranteed.

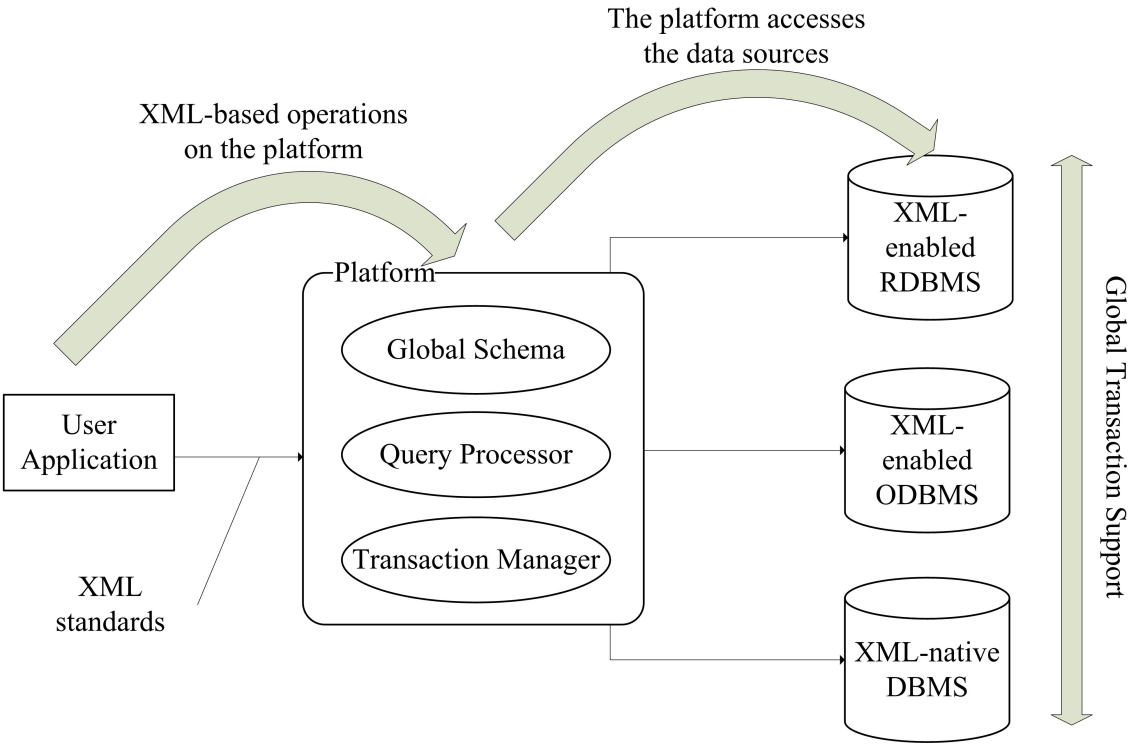

Figure 3.1: Overview of the system.

3.2 System Overview

Technically, the platform is composed of three major components: a distributed query processor (DQP), a distributed transaction manager (DTM), and a distributed schema manager (DSM). Although they are conceptually distinct components, they cooperate together attempting to perform operations in an efficient way. The query processor may use semantics of the global schema to find a good execution plan or to re-write queries in order to reduce the cost of their evaluation and execution. The DTM might rely on the DSM and the DQP with the aim of optimally scheduling distributed transactions. Figure 3.2 shows how the three components fit on the platform.

DQP, DTM, and DSM operate on the top of an infrastructure which is charged of maintaining connections and converting the platform representations into database-specific formats. From a platform's core prospective, each data source has the same data model, the same data definition language, and the same data manipulation lan-

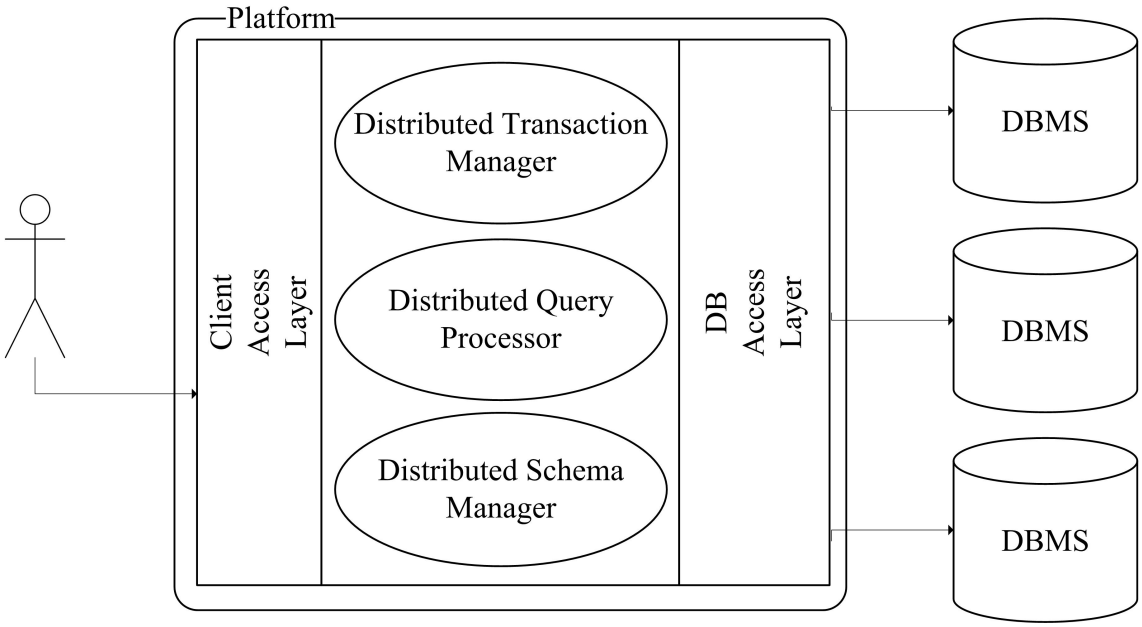

Figure 3.2: Architecture of the platform.

guage. In few words, the core views databases as they would be the same products (a homogeneous system) while, in reality, the system may be composed of several different databases (a heterogeneous system). Particular attention should be paid to the performance aspect due to the massive use of XML parsing. Using an internal XML-based representation is expected to represent a bottleneck on the system, but, after all, this is the cost of building systems providing transparency.

The client access layer provides an interface to external applications. It allows extracting information regarding the global schema, query/modify the multi-database, and define transactions boundaries. Additional features would include the ability to define new virtual schemas and create new instances of the created schemas.

3.3 Requirements Analysis

This section analyses the requirements of the system. This analysis is the basis for the design phase which inspects the main components.

3.3.1 System Requirements

The system requirements are analysed in terms of challenges that have to be met in a distributed system including concurrency, failure handling, transparency, scalability, openness, heterogeneity, security, and finally the eight forms transparency.

Concurrency

A multi-database provides access to shared data. While low level concurrent access is ensured by DBMSs, global atomicity has to be guaranteed by the platform. A priori, the two-phase commit protocol is chosen to coordinate transactions among databases according to the X/Open Distributed Transaction Processing model.

Failure handling

Hardware and software failures can occur at any time at any part of the distributed system. The platform must deal with failures by providing the following functionalities:

- detecting a failure,
- handling the failure (either masking or tolerating),
- and recovering from failures.

Services can be made to tolerate failures by the use of redundant components. However, this project will not analyse the replication of the platform. It is planned to use log files to recover transactions from failures; the report will not provide details regarding recovery procedures since it is estimated that those techniques has been deeply investigated in the past and are still adapted for nowadays systems.

Scalability

A system is described as scalable if will remain effective when there is a significant increase in the number of resources and the number of users [43]. In this context, the number of users performing transactions and the number of databases involved

give an idea of the scale range. For large data repositories and a significant number of users involved, a single platform is not expected to scale but a series of replicated multi-databases would be more appropriate. In this last case, the system could rely on nested or multilevel transactions, or eventually, it might fall into a peer-to-peer category. From a first analysis, the following weaknesses have been identified:

- XML parsing in the core of the platform and elsewhere (XML connectors or core of XML native databases),
- resources being locked by transactions,
- distributed deadlocks,
- and costly queries or queries involving large amount of data.

To scale, the platform have to include efficient algorithms to reduce performance loss. An example could be, as presented in [23] but in a non-distributed environment, applying application level locks mechanisms for increasing concurrent access to DBMSs. Another examples would consist in refining access predicates (i.e. XPath statements) disposing of schema information by attempting to transform ancestor-descendant relationships (denoted by "//") into parent-child relationships (denoted by "/") which, in general, are more efficient when executed.

Openness

Openness is guaranteed by the use of interfaces and standard languages. New database products can be plugged into the platform by implementing a series of interfaces. The adoption of standards such as XQuery permits the infrastructure developer to implement the interfaces with little effort (no query translation required, provided the database supports XQuery). Moreover, the platform is organized according to a layered-based architecture with internal interfaces making it extensible; an inter-layer can be added to support more functionalities or an upper-layer can be built on the top of the existing ones in order to increase the degree of transparency of the system. Those principles are conceived to achieve a so called open distributed system [43].

Heterogeneity

The platform accesses remote databases through database drivers relying on Internet protocols. In addition to this connectivity and because not all database drivers supports distributed transactions protocols, the platform provides a connection with a server operating in-between (i.e. on the top of the database, see section 3.4).

Security

No doubt that strong security is a fundamental requirement for almost any kind of computer system. However, this report will not cover any security aspect since it is out of scope of this research project.

Transparency

Transparency is an important concept in complex computer systems. The ANSA Reference Manual and the International Standards Organization's Reference Model for Open Distributed Processing identify eight forms of transparency (partially covering the same concepts presented in the previous sections):

- Access transparency: local and distant resources are accessible using identical operations through unified query languages and interfaces.
- Location transparency: high-level software components access call operation provided by the underlying infrastructure which will propagate calls to distant resource. As a consequence, application modules do not have knowledge of the location of databases.
- Concurrency transparency: distributed transactions coordinated by the platform's transaction manager are executed in parallel, according to ACID constraints.
- Replication transparency: replication mechanisms could be put in place for making the system fault-tolerant. Replication is not covered in this report.

- Failure transparency: traditional recovery techniques are part of the system to deal with failures.

- Mobility transparency: provided the appropriate driver and the conformity with the global schema, a database can be easily integrated in the multi-database. Configuration files permit to dynamically load deployment-specific parameters. Those expedients allow to quickly moving the database or the platform elsewhere on another machine.

- Performance transparency: clear interfaces among components allow developers to improve or literally replace existing modules for performance purposes. Configurations settings also grant system administrators to tune the system according to the deployment context.

3.3.2 Infrastructure Requirements

The databases have to have some basic requirements for being integrated in the distributed database system. The first, most obvious, requirement is that the DBMS have to support local transactions. A support for the two-phase commit protocol within the driver would avoid writing infrastructure code. Then, it is suitable to define specific requirements such as the locking granularity provided on the DBMS. Ideally, locks should be applied at a node-level and not at the whole document. It would be unacceptable to lock the whole document while a single transaction is performing a modification on it. However, this is not always possible since most of the XML-enabled apply locks to at the document level. Only few systems provide node-level locking, mainly XML native repositories.
Finally, the multi-database could support additional ambitious functionalities including the ability to define save points in case of long transactions or to support schema modifications when a third party modify local schemas. The latter, apart from being a very tough task, requires DBMS to support triggers. Those theoretic features are not covered in this report.
Table 3.1 summarise the basic requirements of database to be integrated on the distributed database system.

Feature	Required	Optional
Java driver	X	
Local transaction support	X	
Distributed transaction support		X
Concurrency control at the element level		X
Savepoints		X
Triggers		X

Table 3.1: Database Requirements

3.4 System Architecture Revisited

Following the requirement's analysis, the system can be seen as two main software aggregates: the main multi-database server, named XML Distributed Database Management Platform (XDDBMP), and, in the case where a database does not dispose of a remote connectivity driver, a second decentralized server. The latter, called XML Data Base Management Extension (XDBME), operates on the top of the database. Figure 3.3 depicts this concept.

XDDBMP contains the three managers mentioned previously (DTM, DQP, and DSM) whereas XDBME encapsulates the decentralized software infrastructure charged to translate the representations used on the core of XDDBMP as well as to provide distributed transactional functionalities. The DTP model briefly introduced in 2.3.2 is used as a reference for the design and implementation of the entire system.

Note that this architecture mainly describes how the software is deployed and that a database can be integrated on the multi-database either using the provided driver (when available e.g. JDBC for Java) or the implemented connectivity using the XML-RPC protocol (see section 3.5.1).

3.5 Design of the Platform Components

This section aims to describe the components introduced in Figure 3.2. For the connectivity infrastructure and the platform components, it is presented the architecture and the services provided to the upper layers. The user access layer is

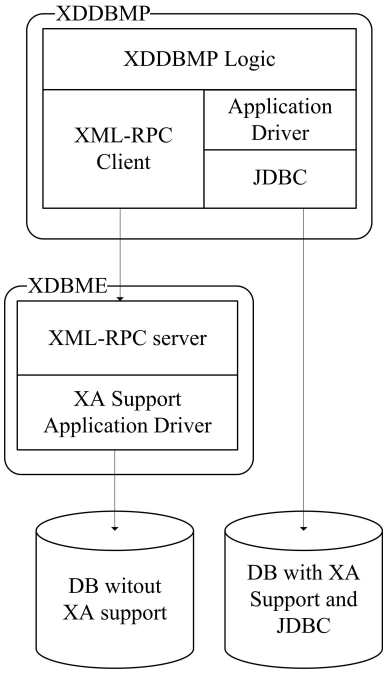

Figure 3.3: Architecture of the platform revisited.

covered in terms of typical application scenarios.

3.5.1 Connectivity Layer

The communication stack incorporating the CRM, the OSI TP, and all its interfaces (see section 2.3.2) is implemented, when possible, following the directives of the X/Open specification. The connectivity layer includes interfaces conceived to provide an homogeneous access to the databases that have to be integrated.

Communication protocol

The XDDBMP communicates with XDBME via an underlying protocol that was selected from the following candidates:

- SOAP (Simple Object Access Protocol)
- XML-RPC (XML Remote Procedure Call)
- A dedicated protocol specifically designed for the application using TCP sockets

All protocols have advantages and disadvantages for this application. SOAP has gained popularity as a protocol for Web Services. A downside of SOAP within this project is that it requires a specific server (e.g. servlet container) to run.
Differently, XML-RPC is a simple interoperable protocol that can be used to implement a server and client from scratch using a library. A drawback of this protocol is that it not suitable to carry complex data structures since it has a limited data type support (not suitable for binary data). Designing a specific protocol using sockets could lead to interoperability problems (serializing objects) in the case where the system has to communicate with other peers.

From the performance prospective, the results presented in [28], which aims to compare various aspect of a client-server implementation using XML-RPC and a typical java client-server socket implementation, shows that the two alternatives achieve the same performance in terms of small request/responses while XML-RPC is slower when transporting large amount of data. Other language-dependent communication technologies implementations such as Java RMI were not considered.
After analysing several aspects of the mentioned protocols, XML-RPC was selected as the communication protocol between XDDBMP and XDBME essentially because of its simplicity, portability, and performance [28].

Data access interfaces

The communication has to provide capabilities to execute read and write statements associated with a transaction branch. The interface below is the Java mapping of the industry standard XA interface based on the X/Open CAE Specification.

```
// javax.transaction.xa.XAResource
void    commit(Xid xid, boolean onePhase);
void    end(Xid xid, int flags);
void    forget(Xid xid);
int     getTransactionTimeout();
boolean isSameRM(XAResource xares);
```

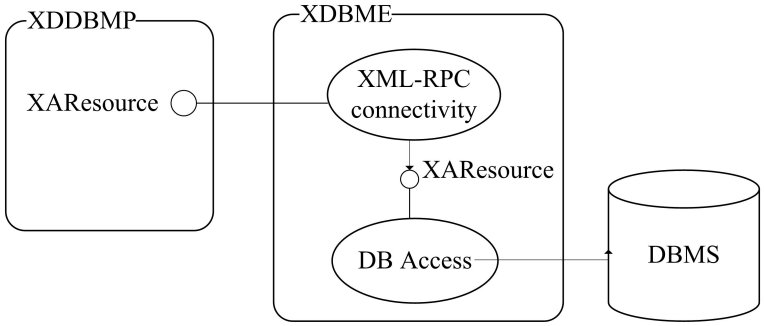

Figure 3.4: XAResource interface on the platform.

```
int     prepare(Xid xid);
Xid[]   recover(int flag);
void    rollback(Xid xid);
boolean setTransactionTimeout(int seconds);
void    start(Xid xid, int flags);
```

In addition to the function required to run the protocol (start/end, prepare, rollback/commit), the interface also includes functions for managing the resource manager (timeouts and recovery). Note the parameter *Xid*, which identifies a global transaction by wrapping both global and branch identifiers in a language-independent object (identifiers are defined through a binary buffer). The XA interface may be present at two different locations on the connectivity layer: in XDDBMP just below the distributed transaction manager, and between XDBME and the resource manager of the DBMS being connected not having an XA support built in the driver. Figure 3.4 shows where the XAResource interface is located on the platform.

The use of XA interfaces in different parts of the platform is conceived to modularize the application, allowing, for instance, to develop an alternative communication stack using Web Services. Furthermore, it is also possible to build an XA support for a DBMS behind XDBME without making changes elsewhere on the software.

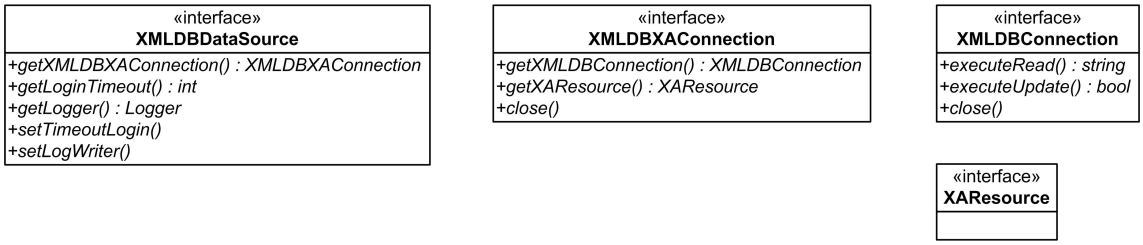

Figure 3.5: Connectivity layer interfaces.

The data access interfaces shown in Figure 3.5 laid the foundations of the connectivity layer. There are three major interfaces: *XMLDBDataSource*, *XMLDBXAConnection*, and *XMLDBConnection*. *XMLDBDataSource* is a sort of factory that, when initiated, provides an *XMLDBXAConnection* that contains XA capabilities. An *XMLDBXAConnection* is able to generate an *XAResource* and an *XMLDBConnection*. An *XMLDBConnection* conform object provides the means to query/modify a database.

Note that in Figure 3.5 the XA interface does not contain the method definitions presented previously. According to the class diagram, an *XMLDBConnection* and an *XAResource* instances are always associated with a *XMLDBXAConnection*. As a consequence, a statement will be executed in a transaction context (*XAResource*). This concept was inspired by the JTA infrastructure (see 2.3.3) including the Connection interface and relative data source factories. By the way, this was designed for performance purposes implementing connection pooling embedded in the database driver.

The layered view in Figure 3.6 presents the overall communication stack. It shows the two communication variants: either a XML-RPC or a socket based connectivity (JDBC). On the top of the stack there are the high level components including the client access layer and the DTM/DQP/DSM modules. The managers are bound to the infrastructure with the data access interfaces. Again, there might be more connectivity alternatives in addition to the XML-RPC and the socked connectivity. The XA support for the XML-RPC connectivity is implemented behind the access interfaces along with the local query processor charged to translate, if necessary,

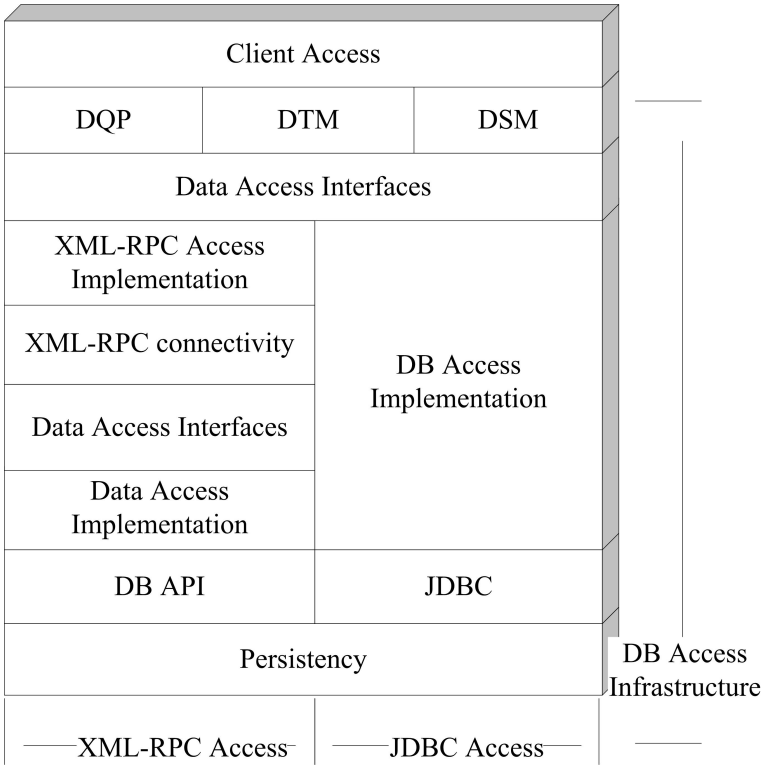

Figure 3.6: A layered view of the platform.

the data manipulation language defined on the platform into that of the database product.

3.5.2 Distributed Query Processor

Query processing involves a series of operations that range from the validation of the syntax to the execution of a query. In a distributed environment the query processing procedure requires few additional steps because a statement must be deconstructed into sub-queries according to directives indicating on which database a sub-query has to be executed. It may also require an arbitrary execution order (i.e. query scheduling) and an optimization of the statement (i.e. query optimization). Distributed query processing will go hand in hand with the schema manager and the transaction manager to validate deconstructed queries and to eventually delegate them to the appropriate databases. The DQP performs the following major operations when treating a query:

1. query validation,

2. query fragmentation/mapping,

3. query optimization,

4. transformation of the query into an internal representation suitable for the DTM,

5. and execution of the query.

Query validation

If at present XQuery would include update operations, it would be the best candidate to use for the platform. Unfortunately this is not the case, but the W3C XML Query Working Group intends to add support for updates in a future version of XQuery (XQuery Update Facility). Since XQuery can also be expressed with an XML representation, known as XQueryX, it is expected that there will be an XML version for XQuery Update too. XQueryX, being an XML-base language, is associated with an XML Schema that can be used for validation. Despite the fact that an XML-based query language may lead to processing overhead comparing to an SQL-like language ('normal' XQuery syntax), XSL could be used to process the transaction definition. No studies have been found comparing the performance of both approaches.
A query/update statement is generally validated by the platform by consulting the global schema it holds, and before being executed, by the DBMS. If a query is not conforming to the global schema, the query is not executed and the global transaction aborted.

Query fragmentation and mapping

Location transparency involves some sort of mappings on the system. Transparency is achieved in a two-layer scheme as depicted in Figure 3.7. The lower layer, logically map a query with the respective database while the upper layer works with the DSM in order to provide more abstraction by omitting any location indication.
The mapping at the lower level is done by using a modified version of the XML

Namespaces Recommendation. XML Namespaces are a W3C standard allowing element type names and attribute names to be qualified with a URI, preventing from confusing two elements that have the same name but different meanings. Although it can be argued that XML Namespaces are confusing (the URI does not point to any resource), it is a sufficient schema to deal with a distributed environment (Web Services also rely on Namespaces). The idea is to identify databases with a namespace-like definition as reported on the above example:

docnamespace=

"xmldb://kdeg.cs.tcd.ie/oracleserver/oracledatabase/collection1/document.xml"

Essentially, the main difference between a standard namespace and the proposed definition is the 'virtual protocol' ('xmldb' instead of 'http') and the fact that part of the namespace gives indication of the database and, to a certain extent, its organization. But not on its location! In the small example above, it is possible to infer that the query will be executed on the database 'oracledatabase', on the collection 'collection1', and on the document 'document1.xml'. In contrast 'kdeg.cs.tcd.ie/oracleserver' is not the real location of the database; it is just a way to encourage the definition of unique namespaces on the multi-database system with a familiar means such as Internet domains. The actual location is specified on a deployment descriptor consulted by the platform when starting up. The definition on the deployment descriptor has to match with that of the queries handled at the lower level of the query processor.

The upper layer simply provides the mapping between the global schema and the internal namespace representation using standard namespaces. Figure 3.7 illustrates the journey of a set of queries through the query analyser module.

The introduced namespace model permits to achieve a unique high-level storage model that is not currently common for XML repositories. One may take for granted the fact that a RDBMS can have essentially several databases including tables. Currently, XML native databases stores data in collections (or containers) while some XML-enabled relational databases stores XML documents in tables or in other entities. For instance, Oracle stores XML documents in a special table

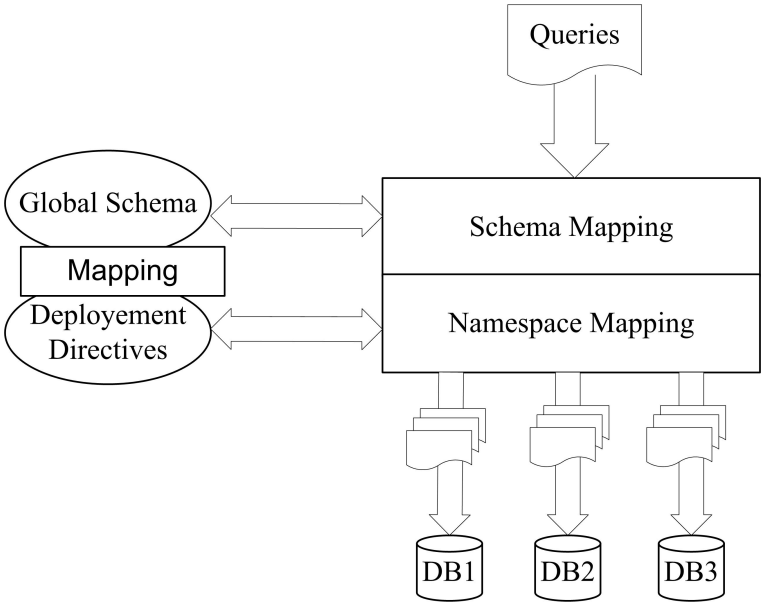

Figure 3.7: Query fragmentation and mapping.

of type 'XMLType'. A unique storage scheme defining the basic storage entities as database, collection, and document allows integrating any XML database that could have a different data organization but, in some way, it can be converted in the common view.

Query optimization

The platform may include a query optimization component acting at a various level. It could operate at a single database level attempting to optimize ready-to-execute statements. Alternatively, it could act at global levels, optimising querying by extracting some semantics from the sequence of operations.
Of course, query optimization and query languages more in general are other challenging research areas that are not be investigated in this project. Section 2.2.3 lists a couple of relevant papers that cover this topic.

Internal representation and execution

The query processor takes the chance to transform the sequence of statements in an internal representation, which is described later on in section 3.6, suitable for further internal processing by the DTP. In fact, as already mentioned, XML pars-

ing is expensive in terms of processing and consequently an important goal is to parse the transaction definition and the queries a single time. The object-oriented internal representation and related functionalities are composed of appropriate data structures to quickly obtain information regarding properties of a query.
The DTM exploits the representation to build an execution plan and eventually execute the queries. An example of a functionality provided by the object representation is the ability to know within a short delay, given two update object statements U1 and U2, whether or not U1 and U2 attempts to update the same node or an ancestor-descendent node. The goal of the described and other functionalities is to speed up the transaction processing step.

3.5.3 Distributed Transaction Manager

The DTP is responsible for designing each global transaction so that it is performed correctly guaranteeing isolation (ACID properties). The DTM handles transactions at the global level and at a sub-transaction level. At a global level, the software module deals with distributed transactions attempting to access data on the remote databases. As the concurrency control is performed on each remote database, the DTM has to control the two-phase commit protocol and the distributed deadlock monitor.

At a sub-transaction level, the DTM can execute statements, when possible, in parallel. For instance, if an update statement of X on DB1 does not depend on a read statement of Y on DB2, then the two statements may be executed concurrently and independently on the respective databases. During this and other analysis, it could be a good opportunity to seek potential deadlock situations in a transaction definition. More generally, distributed deadlock, which might arise as the result of waits imposed by pessimistic approach at a global level. It can be resolved using timeouts, timestamps, or wait-for-graph representations.

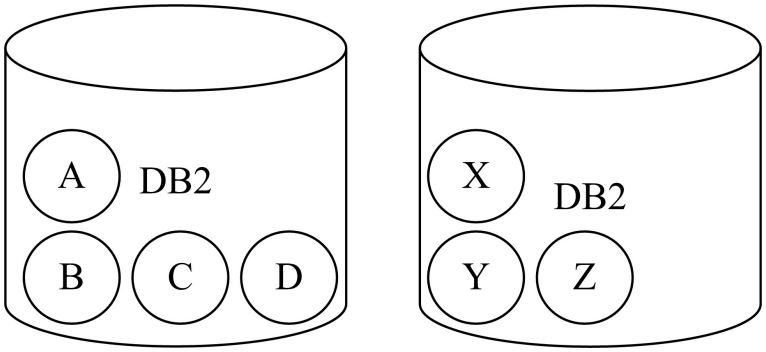

Figure 3.8: Database content scenario.

The transaction component should be developed in a way that it can be easy to change transaction scheduler. Thus, a sequential scheduler will be implemented as a reference for performance experiments. It is also expected to develop a more advance scheduler attempting to run tasks in parallel. The transaction scheduler uses the technique of grouping statements that can be executed in parallel in steps. Let us consider two database, DB1 and DB2, having respectively elements A;B;C and X;Y;Z depicted in Figure 3.8. Now, consider the transaction definition on Table 3.2.

Transaction
write(DB1, read(DB2, X))
read(DB1, B)
read(DB1, C)
read(DB2, Y)
write(DB2, read(DB1, D))

Table 3.2: Case study: a trasaction definition

With a step-based execution approach, the first write operation can be executed in parallel with the dependant read operation but they both have to synchronize at the end in order, for the write operation, to obtain the result of the read statement. This operation could be executed in a sequential way too, since the write statement has to wait for the result of the read statement anyway. The three independent read operations can be executed concurrently (two groups in parallel since two operations are performed on the same database). Finally, the last write statement has to

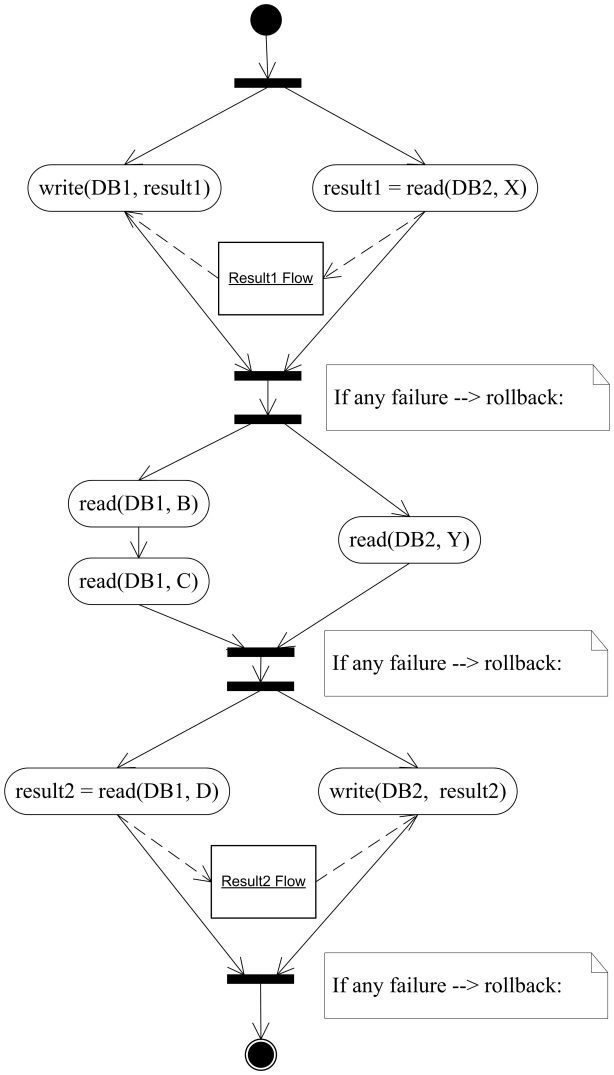

Figure 3.9: Case study - a transaction execution.

synchronize with the read statement exactly like the first one. The execution can be resumed in the activity diagram reported in Figure 3.9.

Apart from the two simple transaction schedulers presented, other techniques can be put in place to develop an alternative transaction scheduler achieving better results. In decades, the literature has produced a remarkable number of mechanisms linked to transaction scheduling.

As the platform may fail, the DTM logs the operations belonging to a distributed transaction that need to be re-done after recovering. The two-phase commit protocol includes guidelines regarding which information has to be persistent after a given operation.

Transaction definition language

XQueryX is XQuery in an XML format. Although is not human-readable, it is particularly suitable for processing (XSL in particular). A suitable data manipulation language for the platform would have an XML syntax where update and query predicate are combined as presented on the case study on Table 3.2. As mentioned on a number of occasions, such a language is not available yet. A possible solution would be to combine XQueryX and XUpdate since they both have XML syntax. A new XML schema definition might extend both schemas so that rules can be defined regarding the allowed operation (e.g. obviously it is only possible to write data that has been read - not vice-versa). Implementing this scheme might require quite a big effort that can be unnecessary since the W3C is expected to produce a new version of XQuery including update capabilities. The implementation phase will assess whether this implementation would be worthless or an available alternative could be adopted instead.

3.5.4 Distributed Schema Manager

The distributed schema manager provides the platform components the services regarding the global schema view over the integrated databases. There could be a passive or an active DSM. In passive DSM, the global schema is defined once for the whole platform life-cycle. In contrast, an active DSM would permit external application to define new schemas derived from local ones and/or, with some limitations, modify an existing global schema view. Web Services on local sites might provide information regarding local schemas so that the platform could offer the means to manage the existing global schema view. Since during the background research in this area any work has been found and this topic might be out of scope of this

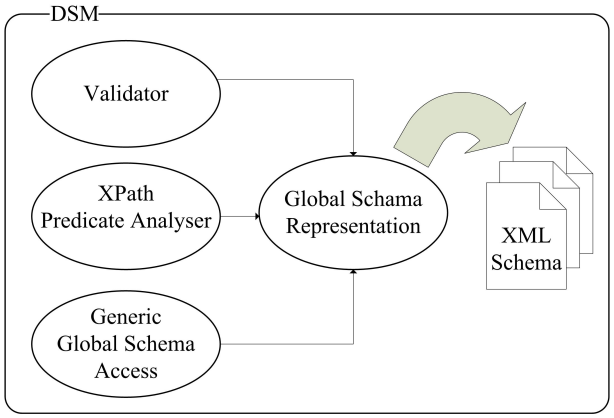

Figure 3.10: DSM Architecture.

project, it is hard to put forward such a complex software component. Because of this, this subsection covers the passive DSM only. It is proposed an overview with its functional components.

The DSM module includes one or more XML Schema(s) representing the global schema view. XML Schemas are stored locally on the file system. When the platform starts up, the schemas are loaded into memory in an internal representation suitable to the software modules that needs to access the information of the schema view. The software modules are, as shown in Figure 3.10, a query validator, an XPath predicate analyser, and a generic global schema access.

The query validator receives statements that have to be validated against the global schema. Whenever a query or modification predicate is not valid, the platform can return an error to the application client without performing further processing (the execution will fail anyway).

The XPath predicate analyser provides functionalities to the query processor for optimization purposes. Thus, an XPath predicate is analysed and, when applicable, processed taking into account the semantics of the schema definition. In the best case scenario, the component is able to return the query processor an equivalent XPath expression that is more efficient when executed. In particular, transforming

ancestor-descendant relationships ("//") into parent-child relationships ("/") may significantly reduce the response time when executing the query.

The schema access point is designed for out-platform accesses using XQuery. Intelligent client applications may want to find out the information that a multi-database can potentially store from its schema, and if necessary, query it. This last component may be suitable to use in conjunction with Web Services acting on the client access layer.

3.5.5 Client Access Layer

The client access layer may provide different kinds of access according to the requirements of applications. A typical access would be a server permitting client applications to query/update the multi-database performing transactions. A server could provide a Web view of the information contained in the platform. In this case, the platform might be directly embedded into the Web application. Alternatively, a server supporting SOAP can provide Web Services to remote applications that can perform transactions using WS-specific transaction processing definitions.
In both cases, some kinds of access control have to be put in place, commonly required by every information system.

3.6 Interaction between DTM and DQP

In order to understand the interaction between the distributed transaction manager and the distributed query processor presented respectively in section 3.5.3 and 3.5.2, let us consider a user application willing to execute a transaction on the platform. The main classes involved are included in the class diagram shown in Figure 3.11. Those classes are in fact the core of the platform. They partially include what has been referred previously as the internal representation.

The class *XDDBMPlatform* is the main class representing the platform. The instance of this class, instantiated when the platform starts up, reads the deployment

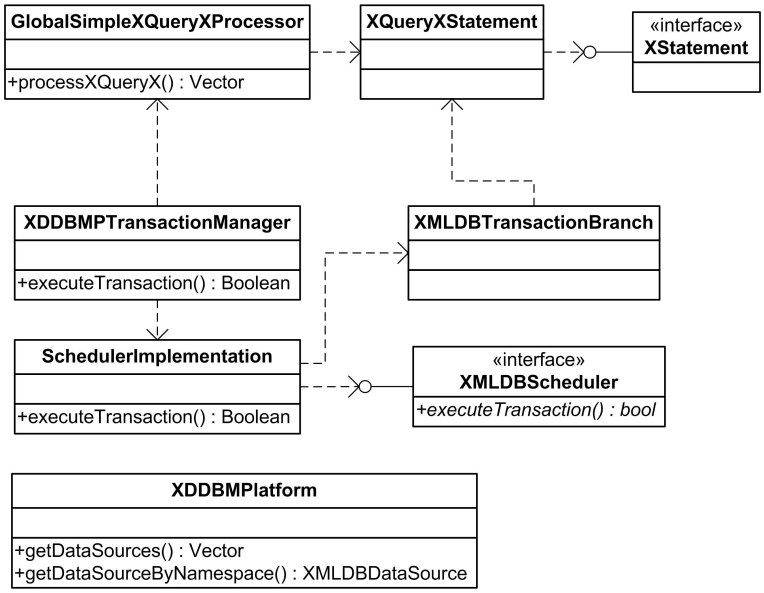

Figure 3.11: Class diagram of the platform's core.

directives in a configuration file and creates a connection to the remote DBMS.

An instance of class *XDDBMPTransactionManager*, created for each distributed transaction that has to be executed, exploits the connections (identified by a namespace) created to the data sources to handle a transaction definition. The class *XDDBMPTransactionManager* can be considered the coordinator of the whole process which is resumed in the sequence diagram reported in Figure 3.12.

Once that an object *XDDBMPTransactionManager* has been created by a client manager that has received the transaction by a client application, it invokes a method of a DQP component class, *GlobalSimpleXQueryXProcessor*, for processing the transaction definition. Basically, the function *processXQueryX* of *GlobalSimpleXQueryXProcessor* fragmentate the global transaction in a set of single statements represented by an object conform to the interface *XStatement* (*SimpleXQueryXStatement* in this implementation). The statements are sequentially stored in a object array and returned to the transaction manager (*XDDBMPTransactionManager*).

At this stage, the coordinator is ready to call the scheduler passing as a parameter the list of statements to be executed. The implementation of the scheduler may vary but it has to implement the interface *XMLDBScheduler*.

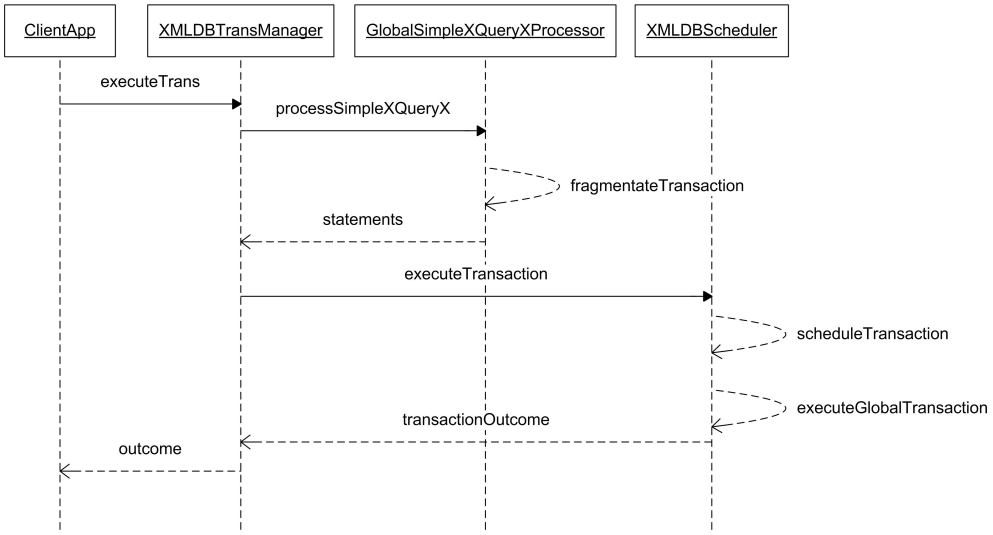

Figure 3.12: Sequence diagram of a transaction execution.

Depending on the type of scheduler, the list of query/update predicates are analysed and put in an object representing a transaction branch (*XMLDBTransaction-Branch*). There are as many transaction branch objects as the number of databases involved in the distributed transaction. Meanwhile, the transaction scheduler can start the two-phase commit protocol. The query/update predicates are passed to the under-lying infrastructure as they will be translated later on.

Finally, the transaction manager receives the final outcome from the scheduler and returns it to the client. Note that for space reasons, the class diagram does not include all methods and attributes and the sequence diagram does not reflect a detailed sequence of calls.

3.7 Design Issues

Again, the main design issue was related to the lack of a standard language to query/update XML documents. If a standard data manipulation language had been available, it would have been strongly possible that some existing open-source stand-alone query processors could have been integrated onto the platform. Since this was not the case, this was built from scratch resulting in a limited operation set.

A more technical design issue involved the duality of the resource manager including the local transaction manager and the query interface. The DTP model specifies the interfaces that have to be used to coordinate a distributed transaction. But the model does not say anything regarding how a query predicate can be executed in a sub-transaction context. Hence, in order to build the XML-RPC connectivity, it was required to come up with a mechanism devised to assign a statement within a transaction. This mechanism basically relies on logical connections that the coordinator creates for each resource manager involved in a transaction. This allows having a logical connection for each transaction branch with the relative set of queries to be executed.

Chapter 4

Implementation

This chapter describes a partial implementation of the system presented in Chapter 3. It introduces the deployment scheme, the databases, and software libraries that have been used. Based on the deployment scheme, it is provided a description of the components with related technologies, algorithms, and the objects involved.

4.1 Implementation Overview

Figure 4.1 introduces the implementation of the platform. Basic versions of the distributed transaction manager and the distributed query processor have been implemented. They rely on the data access infrastructure that has been built for two databases (the databases are presented in section 4.2). For the 'Oracle XML DB' database, the platform uses a standard Java access using sockets (JDBC). For the database 'SleepyCat XML DB', an XML-RPC connectivity has been implemented. In addition, because few databases support global transactions, a basic XA support has been created. The distributed schema manager component has not been implemented.

4.2 Databases

The prototype of the platform has been built with two XML databases: Oracle XML DB and SleepyCat Berkeley DB XML. Both databases can be obtained free

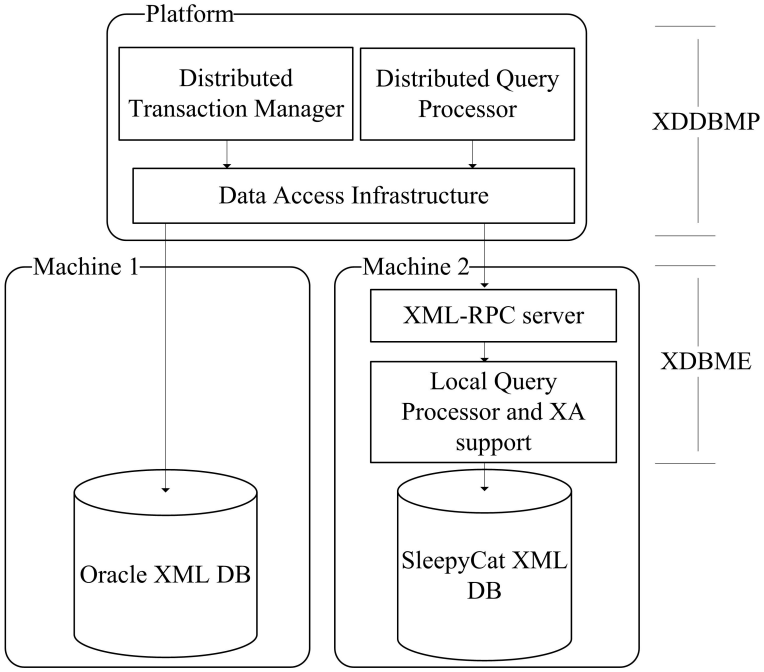

Figure 4.1: Implementation overview.

of charge for non-commercial use. Oracle XML DB is a XML-enabled database whereas SleepyCat Berkeley DB XML is considered an XML native database. The last, however, applies locks at the document level. At least a native XML repository with a locking node-level mechanism would have been more appropriate, but either open-source databases do not support transactions or those which do provide support are commercial products. Table 4.1 reports the databases that have been considered.

Database	**Licence**	**Transaction support**	**XA support**
Apache Xindice	open-source	no	no
Berkeley DB XML	open-source	yes	no
dbXML [18]	open-source	partial	no
eXist	open-source	no	no
Oracle XML DB	commercial	yes	yes
Tamino XML Server	commercial	yes	no

Table 4.1: Surveyed databases

Oracle XML DB

Oracle XML DB adds native storage and retrieval of XML content to the capabilities of Oracle Database 10g release 2. The XML engine relies on the vendor-specific object XMLType representing an XML document on the database internal storage format. This storage format provides what is known as DOM Fidelity, which guarantees that none of the information contained in the XML document is lost when the document is stored in the Oracle database. The XMLType data type can be used like other SQL data types, including in table definitions, view definitions and as a PL/SQL variable, parameter or return type via an SQL-like language. The DBMS provides access via various data manipulation languages including XPath, XQuery, DOM; it fully support the XML Schema recommendation.

Oracle XMLDB provides two options for storing XML in the database. In the first, data is stored in the unstructured mode which uses the Character Large Object (CLOB) data type to persist the XML document as a string of bytes in the database. The second is the structured storage involving shredding the XML document in a set of SQL objects and tables. Structured storage is only available when the XML conforms to an XML schema.

Since the distributed schema manager has not been implemented, data is stored in an unstructured way, but following a data non-implicit schema definition (i.e. XML Schema in not present on the database but the documents and the query performed will not violate the constraints).

SleepyCat Berkeley DB XML

Berkeley DB XML is a native XML database engine providing XQuery access into document containers. XML documents are stored and indexed in their native format using Berkeley DB as the transactional database engine. This product is not a client/server database management system; it is a C++ library linked directly into an application. The aim is avoiding client server network overhead.

The product has been chosen because it one among few open-source XML databases supporting transactions and XQuery. Finally, in addition to the in-built deadlock

detector, the database provides a deadlock detector the developer can invoke though the access API.

4.3 Libraries

A number of libraries and third-party software have been used to develop the platform. This section briefly introduces each of them.

XML-RPC

XML-RPC, which is a protocol that uses XML over HTTP implementing remote procedure calls, is the communication protocol used between XDDBMP and XDBME. The Apache Web Service Project, which is part of the Apache Software Foundation, provides a Java implementation of the protocol allowing implementing both client and server. The library provides the means to develop a simple client/server communication or more advanced ones such as the ability to support a simple authentication service.

XML Processing

The Apache XML Project, another project of the Apache Software Foundation, plays an important role in the development of XML technologies providing, among other things, XML processors.
Xerces provides XML parsing for both Java and C++. It implements the W3C XML and DOM, as well as the de facto SAX standard. Initial support for XML Schema is also provided.
Xalan provides an XSLT stylesheet processor. Xalan fully implements the W3C XSLT and XPath recommendations. The library also includes an XPath Processor that can be used as a stand-alone unit. Both Xerces and Xalan are used in this project.

Logging

Although the Java standard library disposes of a basic log library, a third-party library has been adopted. The Apache Project provides a Java library, named log4j designed for logging. It features several interesting functionalities for the development of flexible applications. In particular, it allows defining configuration settings on files managing the logger behaviour. Additionally, it supports several kinds of output formats such as text, XML, or HTML.

Concurrent programming

The development of software entities such as a transaction scheduler requires concurrent programming utilities. The Java standard library has basic utilities like semaphores and more high level ones (the lasts only from release of J2SE 5.0). In fact, from J2SE5 the java.util.concurrent package includes exclusion, resource management, and related synchronization aids, previously provided by Oswego State University of New York. To avoid requiring the platform running exclusively with version J2SE 5.0, the source code has been obtained and integrated in the platform code. The transaction scheduler uses Barriers (Rendezvous) to synchronize groups of threads performing a transaction.

4.4 XDDBMP

When starting up, XDDBMP reads a deployment configuration file indicating the available databases and their namespaces. After connecting to the DBMS using these settings, it is ready for handling transaction requests. Each new transaction requires the creation of a transaction manager and a query processor instance that is charged to perform the transaction and return the result/outcome.

4.4.1 Oracle and XML-RPC Client Side Connectivity

The connectivity for the Oracle database consists of an implementation of the data access interfaces presented in section 3.5.1 using JDBC and an XSL script to trans-

form statements of the global definition into the Oracle syntax. An example of this transformation including a fragment of the XSL program is reported in Appendix B.1. For performance reasons, the XSL script is loaded into memory once the object representing the data source is created. The implementation of the Oracle connectivity is relatively straightforward since the resource creation using JDBC almost match that adopted for the data access interfaces. So, the implementation looks more like a wrapper of the Oracle JDBC objects than a 'from-scratch' implementation.

The XML-RPC client side connectivity requires more code. As for Oracle, it implements the data access interfaces providing an end-to-end connectivity to the resource manager of the remote database. The client side module mainly serializes and deserializes objects, queries, and results of queries. This module is also charged to handle exceptions propagate them to the platform.

4.4.2 Distributed Query Processor

The DQP implements the lower layer presented in section 3.5.2. This component receives a transaction definition containing the queries that have to be executed. It has been decided to create a native XML syntax, called SimpleXQueryX, which is a subset of the set of operations that can be executed on XML documents. Not all operations defined have been realized for both databases. Oracle supports *update*, *read*, and *delete* operations. SleepyCat supports almost all operation expect for *readnode*. However, the DTP considers in its logic the operations *readelement* and *update* only.
Table 4.2 reports defined operations and the fragment of code below shows a small example of an insertion of a node read from a different database. An example of SimpleXQueryX is reported in Appendix A.

```
<append docnamespace="xmldb://db1.ie/db1/col1/doc1.xml"
    select="/element">
        <readnode docnamespace="xmldb://db2.ie/db1/col1/doc2.xml"
```

```
        select="/element/sube[1]"/>
</append>
```

Update operations on nodes (See XUpdate working draft)
insert-before
insert-after
append
update
remove
rename
Inserts
element
attribute
text
Read operations
readelement
readnode

Table 4.2: Implemented data manipulation operations

Using SAX, the query processor parses the definition and creates an internal representation (see section 3.6) composed of several data structures containing single statements converted into an object conform to the interface *XStatement*. This interface fully represents a generic single statement providing information to the components regarding the statement's characteristics. Objects implementing *XStatements* may be encapsulated and chained in order to fully capture nested queries (e.g. write a value on DB1 that has been read from DB2).

Because of the necessity to use the XML processor a little as possible, the DQP parses the transaction definition only once. The result of an execution has been built in the form of a boolean in the case of an update and a string in the case of a read. This is a simplistic approach that can be further extended in a more complete result set that may include additional information. For instance, requests sent to the Tamino Database product return a response described using an XML syntax. Knowing the response schema, it is possible to interpret the answer and convert it to the format required by the application clients.

Distributed Transaction Manager

The DTM takes advantage of the internal representation of a transaction that has been created by the DQP. The DTP performs the two-phase commit protocol using the infrastructure resources created at the XDDBMP start up. Thus, for each transaction, the DTM performs the followings tasks:

1. for each statement
   (a) opens the corresponding database resource manager (if not already opened)
   (b) starts the transaction branch (if not already started)
   (c) inserts the statement to the execution plan data structure (depends on the scheduler being used)
2. executes the statements (also depends on the scheduler being used)
3. for each resource manager opened (i.e. for each database involved in the transaction)
   (a) ends the transaction branch
   (b) prepares the transaction branch and gets the answer from the resource manager
4. for each resource manager opened
   (a) either commits or aborts the transaction branch according to the global outcome
   (b) closes all connections and resources

This procedure is common for each available scheduler which has to implement the interface *XMLDBTransactionSchduler*. This entitles the DTM to easily change the transaction manager, even at runtime depending on the characteristics of the transaction. As presented in the design of this component (section 3.5.3), two transaction schedulers has been implemented: a native one that executes statements as

they are declared in the transaction and a more sophisticated one which tries to execute statements in parallel on different databases.
The simple scheduler is quite straightforward. In contrast, the advanced one is a bit more complex. It essentially uses barriers to synchronize concurrent executions in order to exchange intermediate results, if any, and partial outcomes. Partial outcomes are useful since if there is a failure during the transaction, the whole transaction can be aborted before executing all remaining statements. In general, a set of statements can be executed in parallel if they do not depend of other operations. If there is a write operation depending on a read query, the transaction manager execute both concurrently and eventually, the thread executing the read statement passes the result to the thread holding the write operation. This happens when they both reach their common synchronization point. More efficient are those sequences of independent statements that can be executed in parallel. Technical details regarding the implementation of the scheduler are presented in Appendix C.

4.5 XDBME

XDBME represents the resource manager accessible through XML-RPC. The server has a configuration file on which it is defined the classes and drivers it has to use to operate.

4.5.1 XML-RPC Server Side Connectivity

The XML-RPC server needs to register a handler so that it can process queries sent by the platform. The handler is independent from the database implementation since it accesses the XA connectivity via the data access interfaces. The handler deserializes objects arriving form the platform, calls operations of the resource manager, and returns the outcomes of the execution.
During the implementation it emerged that it was not possible to create new connections to the resource manager to perform each transaction as firstly though in the design phase. Instead, there was the need to implement logical connections pipelined through a physical connection.

The handler operations are those provided by the *XAResource* interface plus the functionalities to execute the queries. The server on which the handler is registered is developed with the Apache XML-RPC Java library.

4.5.2 SleepyCat Binding

Considering the database supports only local transactions, a minimal XA support had to be implemented.

The resource manager implemented for SleepyCat holds an internal data structure running local transactions according to the two-phase commit protocol remotely coordinated by XDDBMP through XML-RPC and XA interfaces. Hence, starting a transaction is related to create a local transaction. Ending and preparing require leave a persistent trace of events (e.g. writing logs) and return to the coordinator the outcome of the transaction branch. Finally, the local resource manager waits for directives whether it has to commit or rollback.

Those services are provided using an object *SleepyCatTransactionBranch* (this class implements the interface *XMLDBXATransactionBranch*) that encapsulates all resources being used. The server handler holds as many *XMLDBXATransactionBranch* objects as the number of active transactions on the local resource manager. The interface allows the server to invoke the reource manager of a database independently from its implementation. This abstraction requires the infrastructure connector written for SleepyCat following the directives of the data access interfaces depicted in Figure 3.5.

The local query processor is charged to execute query/update predicates coming from the platform. To do so, this software module has to analyse the XML query definition (SimpleXQueryX) and execute the corresponding statement. An example of a step of this process is explained in Appendix B.2.

4.6 Platform utilities

The software includes a series of tools in support to the procedures that has to be performed at run time.

Configuration Readers

A dynamic class loader is used for both XDDBMP and XDBME. The tool relies on the Java 'reflect' package (java.lang.reflect) for dynamically creating object depending on the configuration directives of a XML configuration file. It uses SAX to analyse the XML document and it creates the objects needed by the platform. In particular, this is used to create a XMLDBDataSource-like object as shown in the above fragment of the XDDBMP configuration file for Oracle:.

```
...

<dbaccess>
  <accessclass>
    ie.tcd.xmldb.xddbmp.driver.oracle.OracleXMLDBDataSource
  </accessclass>
  <classparameters>
    <dbhost>kdeg-uml-9.cs.tcd.ie</dbhost>
    <dbport>1521</dbport>
    <dbalias>oraclexmldb</dbalias>
    <dbcontainer>orcl</dbcontainer>
    <username>scott</username>
    <password>tiger</password>
  </classparameters>
</dbaccess>
...
```

The number of arguments has to match that of constructor of the object to be loaded. The configurations setting on the example allows to create an OracleXMLDBDataSource that will be used by the platforms to access the remote Oracle resource manager.

Logging

Since system recovery has not been implemented, logging has primarily been used for debugging the system. A generic object charged to generate log files (*XMLD-*

BLogger) can be used by every object in the platform. To do so, the object must inherit from the super-class indicating the name of the log file being generated and on which class the logger have to be associated. Typically, *XMLDBDataSources*-conform objects are an extension of the logger object class in order to log the operations performed on the databases.

XPath parser

An XPath parser was created to compare two XPath predicate. This tool was designed to be used by the distributed transaction manager to assess whether two predicates attempt to access the same node or descendant-node. Since the system does not feature the distributed schema manager, the XPath parser does not support ancestor-descendant expressions. In fact, it is impossible without schema indication if, for instance, the predicate //animal[name=tiger] is an ancestor of the node matching the predicate //*[country = India].

4.7 Implementation Issues

Most of the implementation issues were related to the development of the data access infrastructure. The design issue presented in section 3.7 was fully understood once the basic infrastructure was in place. At this stage, a major refactoring had to be carried out in order to introduce the logical connection model.

Other implementation problems were encountered when processing XML data. Although parsing is normally related to a defined schema, there is usually the need to write code that process data which is reusable in different parts of the software. This task is not always simple as it might seem, mostly because it is difficult to write generic functions that process fragments of XML document having different schemas.

Chapter 5

Evaluation

This chapter presents some considerations regarding benchmarking XML databases and an evaluation of the platform consisting two experiments.

5.1 Benchmarking XML Databases

Since XML databases do not have standardized interfaces and query languages yet, it becomes quite difficult designing appropriate performance experiments. Along with the development of XML databases, several benchmarks, such as XMark [39], were also proposed. XMark is intended to help both developers and users to compare XML databases via a set of queries each of which is intended to challenge a particular aspect of the query processor or storage engine.

[38] reports that no studies were found on using benchmarks that can provide users with insights on the impacts of a variety of storage models on XML query performance. The authors propose a set of results on benchmarking on a set of XML database implementations using two XML benchmarks models. [37] argues benchmarks are distinguished in two main variants depending on how the data is stored on the database: schema-less or conform to schema. It is pointed out that making this first distinction would allow to assess the performance impact when having schema support.

In presence of transactions, [23] carried out an evaluation on the system built on the top of a database comparing response times and throughput of client request transactions performed directly on the database or through a middleware.

5.2 Experiments Overview

Database benchmarks are used to assess the overall system performance with a series of test designed to evaluate different aspect of a database management systems. This is the case on the condition that the database fully supports the standard interfaces. The implemented software does not support all the operation sets defined for XML documents. For this reason, a standard benchmark can not be used. Thus, this evaluation considers two experiments specifically designed to test the operations and features supported by the platform.
The evaluation is composed of two experiments. The first is designed to provide a general evaluation of the system whereas the latter aims to compare the scheduler that has been realised. As shown in Figure 5.1, the databases have been populated with three documents (see Appendix D), two on Oracle and one on SleepyCat.

Experiment setup

For the experiments, the platform runs on a Dell Latitude D400 (Intel Pentium M 1600MHz and 256MB of RAM); the operating system is Linux Fedora Core 3. The DBMSs, Oracle and SleepyCat, are installed on a remote machine running Linux Debian. It is a Linux virtual machine (User-mode Linux Kernel) on a machine having a Pentium 4, 3.2GHz of CPU, and 1GB of RAM).

5.3 Global Evaluation

This experiment aims to compare the response time of the platform when executing a set of transactions and the response time of the same transactions executed directly on the databases. To do so, Oracle XML DB is used because it supports XA.

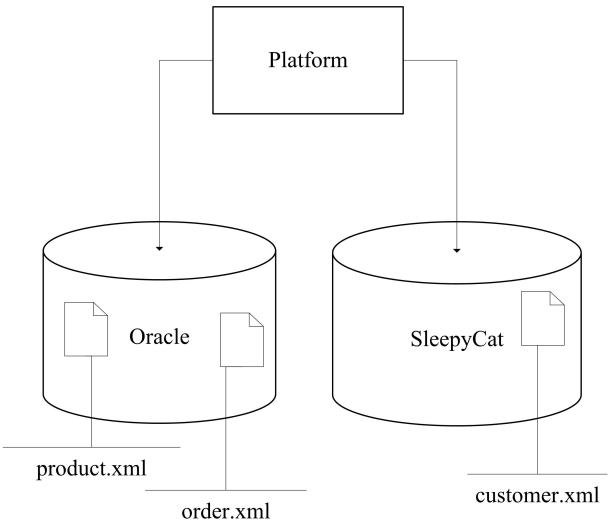

Figure 5.1: Documents stored on the databases for the evaluation.

Four sets of transactions containing around four statements each have been defined. The same transactions have been written in a Java script running them using the XA support of the Oracle JDBC driver.

Initially, the idea was to create two distinct XA connections and executing a distributed transaction emulating two databases while in reality they are applied on the same database (but two distinct XMLType tables).

After testing this experiment scenario, it has emerged an 'abnormal' behaviour: even if in both transaction branches there was an update statement, on one branch the resource manager committed the transaction in advance since it claimed there were read-only statements.

This happened when calling the resource manager asking to enter in a prepared state. The resource manager returned a value meaning that the transaction work has been prepared normally on one branch (all right) and a return value indicating that the transaction branch has been performed in read-only mode and, as a consequence, it had already committed! No documentation was found reporting this restriction; probably because it is unlikely that in a real application one would like to run a distributed transaction on the same database.

The four transactions used are reported in Table 5.1. They are run sequentially for ten times each.

T1
read(/order[1]/quantity)
read(product[1]/price)
update(/product[2]/name, 'DVD')
update(/order[2]/quantity, '550')
read(/order[3]/quantity)
T2
update(/product[1]/name, 'PC1')
update(/order[1]/item, 'PC1')
read(/order[3]/item)
read(/product[2]/name)
T3
read(/order[1]/quantity)
update(/order[2]/price, '995')
read(/product[1]/price)
update(/product[2]/name, 'Printer')
T4
read(/order[1]/quantity)
update(/product[1]/name, 'Laptop')
read(/product[1]/price)
update(/order[1]/quantity, '360')
read(/order[2]/quantity)

Table 5.1: Global Evaluation Transactions

Expected Results

The response times of the transactions performed through the platform is expected to be grater since:

- the query processor has to analyse the queries defined in XML,
- XSL transformations are required for converting statement into the Oracle syntax,
- the transaction manager has to make several decisions when processing the transaction definition,
- and, last but not least, generating log files takes time.

On the other hand, the Java script running the XA transaction is hard coded (no XML processing, no test statements, no loops, etc.). So, theoretically, the script

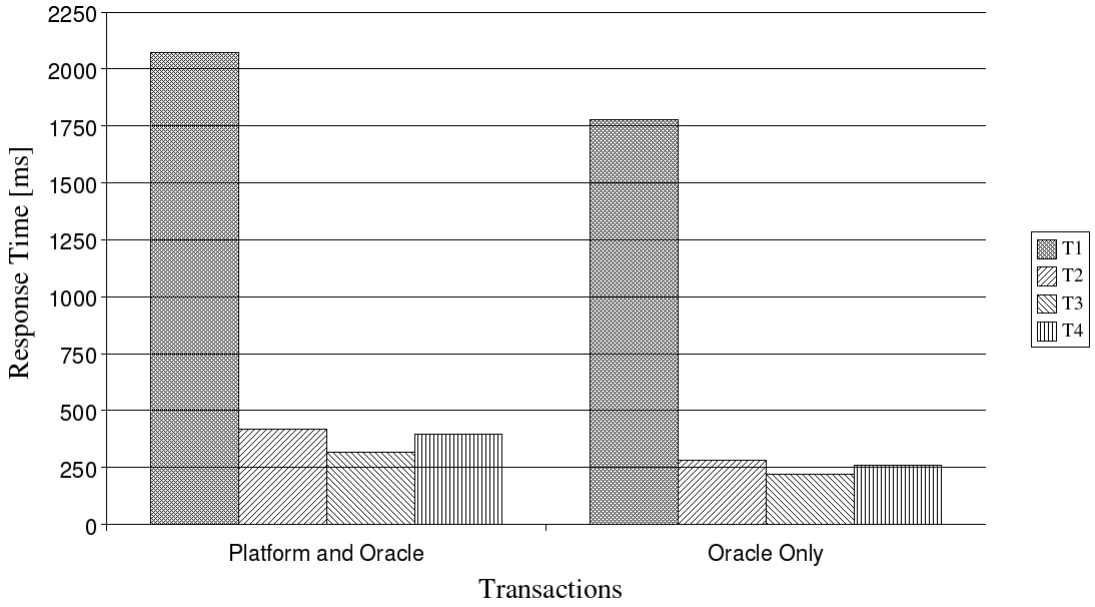

Figure 5.2: Response time comparison.

that runs the transactions can be considered the fastest way using any infrastructure relying on the Oracle JDBC driver. This global system's evaluation aims to quantify the difference between the two response times measured for both scenarios.

Results

As mentioned, the transactions have been run ten times in a sequential way (T1, T2, T3, and T4) for both scenarios. Response time intervals, expressed in milliseconds, have been averaged for each transaction and reported in Figure 5.2.

Discussion

The first noticeable data, on both cases, is the difference of the time required to perform T1. This is probably due to the fact that T1 needs to initiate a physical network connection while T2, T3 and T4 exploit the created connection by communicating with the DB resource manager through logical connections. This is part of the internal implementation of the Oracle JDBC driver attempting to optimise remote resources binding.

Taking into account T2/T2/T3 and using the response time measured directly on Oracle, it emerges a response time for the platform greater than respectively 48%, 43%, and 52%.
It can therefore be analysed that, in presence of simple statements acting on leaf nodes, the platform would require roughly 50% more time to carry out a transaction than an application using the same low-level access technology (JDBC). This might seem a big difference but considering the processing involved supporting a unique manipulation language and the fact that some optimizations could be applied, this is an acceptable result.
Again, a more complete set of operations (e.g. add more nodes) would provide a broader global evaluation investigating, for instance, the impact on performance depending on the kinds of supported operations (e.g. replace many nodes) or the number of requests performed in a transaction.

5.4 Transaction scheduler comparison

The goal of the second experiment is to compare the two transaction schedulers that have been implemented. The simplest scheduler, nicknamed scheduler A, executes statements in the same order as they are defined in the transaction. This option should provide, a priori, a low transaction throughput. The other scheduler, scheduler B, has the ability to execute, when possible, statements in parallel. The experiments consider two transactions, T1 and T2. As reported in Table 5.2. T1 has got statements that do depend on each other while T2 has statements which can be executed independently.
Both transactions have been executed consecutively fifty times. The first execution has been omitted from the measurement for the reasons reported on the previous experiment.

Expected Results

Executing tasks in parallel is generally more efficient unless the tasks are so simple that the effort for creating the resources required for the concurrent execution is not

T1
update(DB1, /products/product[1]/name,
read(DB2, /customers/customer[1]/orders/item[1))
update(DB1, /orders/order[4]/item,
read(DB2, /customers/customer[1]/orders/item[1))
update(DB2, /customers/customer[1]/phone, '0791235678')
read(DB1, /orders/order[1]/quantity)
read(DB1, /orders/order[1]/quantity)
update(DB2, /customers/customer[4]/item[1], 'DVD')
T2
update(DB2, /customers/customer[1]/phone, '011214439')
read(DB2, /orders/order[1]/quantity)
read(DB2, /orders/order[1]/quantity)
read(DB2, /orders/order[1]/quantity)
update(DB2, /customers/customer[4]/orders/item[1], 'DVD')
update(DB1, /products/product[1]/name, 'Laptop')
update(DB1, /orders/order[3]/item, 'DVD')

Table 5.2: Transaction definition for the schedulers comparison

convenient. This could be the case since the creation of a thread to execute a simple read predicate might be pointless. However, for complex operation on a database, concurrent execution can get many benefits to the overall system performance. So, in this case, scheduler B might be more efficient but of the same order of magnitude of the simple one.

Results

The plot in Figure 5.3 reports the results of the experiment. The response time intervals measured for the fifty executions have been averaged for each transaction and for each scheduler.

Discussion

This experiments reveals that scheduler B badly performs (the response time is around 10% greater) comparing to scheduler A for both transactions, when executing simple read and write statements.

It can however be observed the slight difference between the response time needed

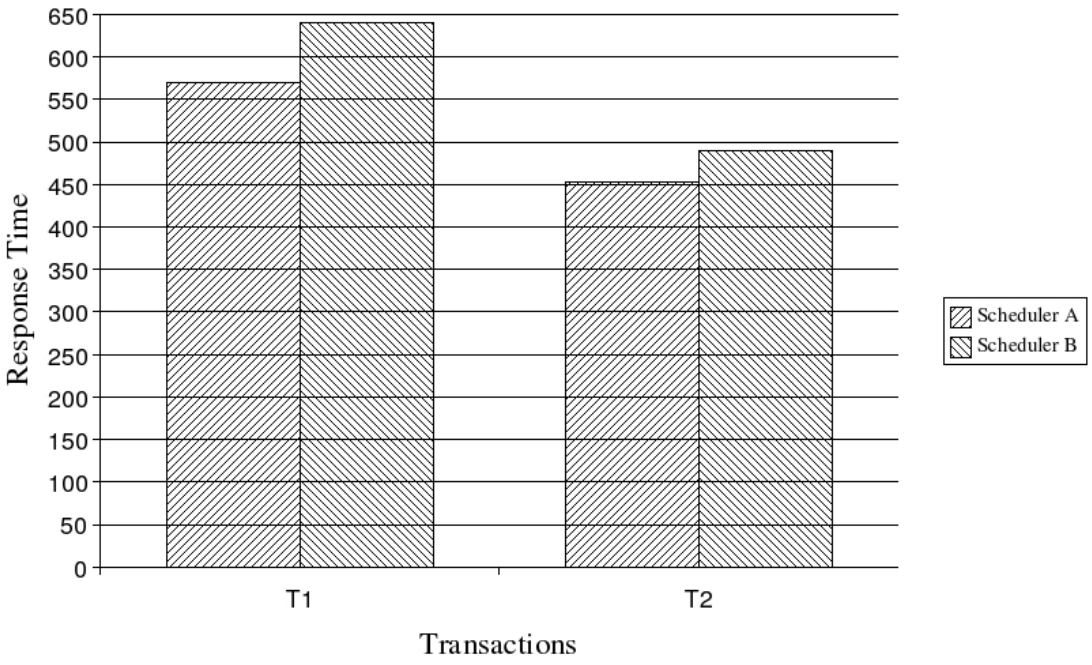

Figure 5.3: Response time for each schedulers.

to execute T1 and T2 of 0.3% (T1 and T2 are respectively 11% and 8% faster with scheduler 1). This could be explained by the fact that the statements of T1 can be fully executed in parallel on both databases.

On one hand, a possible solution to achieve better results would be trying to optimize the management of system resources such as adopting a pool of ready to use thread. On the other hand, scheduler B could be more suitable, as already mentioned, for more complex statements requiring a longer execution time. Alternatively, a third transaction scheduler could be developed using a hybrid approach. Assuming the DTM can rank the cost of the various queries, it could decide the appropriate scheduler to use. Only a set of experiments with a complete range of data manipulation operations would support those hypotheses.

5.5 General Discussion

The first experiment provides an antecedent evaluation of the impact on performance that the layers operating over the database driver have. The latter compares the characteristics of the two implemented transactions schedulers.

As already mentioned, in order to provide a broader evaluation, more data manipulation operations should be supported and experiments have to be carried out measuring the behaviour of the query processor and the transaction schedulers. Additionally, provided a distributed deadlock manager, it would be possible to run the transactions concurrently in order to observe the platform behaviour in a real application scenario. As a matter of fact, locking mechanisms of the adopted databases still provide a poor concurrent access deteriorating the transaction throughput of the platform. After few tests, it has been noticed a high number of collisions resulting in deadlocks and/or in a large number of transactions being aborted. A fully native XML storage with a locking mechanism at the node-level would be more appropriate for this kind of test.

From the platform perspective, a non-native XML database can be integrated but the developer must be aware that if the concurrent access does not provide a sufficient granularity, the whole system performance is compromised. This leads to the supposition that, if the popularity of XML continues to grow, either XML-enabled databases improve their concurrent access on their XML-based facilities or XML-native databases become the first choice when XML data has to be stored and manipulated with transactions.

Chapter 6

Future Work and Conclusions

6.1 Future Work

It is expected that in future the W3C will produce a new XQuery recommendation including update facilities. Then, gradually, database products will support the new standard just as it happened for the current XQuery release. One of the issues encountered in this project, was just the variety of data manipulation languages supported by databases. It is not easy, in these conditions, designing a system using a unique data access language. Having a language including both query and update facilities it would be possible to design and implement or adopt a complete query processor within the platform. In this case, some experiment can be done taking into account different test scenarios. One could be a transaction throughput comparison between XML-enabled databases and native ones. Another could measure the transaction throughput when executing a wide range of operations types.

A further work would consist in investigating how schema semantics expressed thought XML Schemas could be used to optimize operations within an XML multidatabase system. A transaction manager might use this information to detect deadlocks or to improve a transaction schedule. A query processor could consult XML Schema information to optimize queries.

A version of the proposed Distributed Schema Manager could be implemented to assess its feasibility and its limitations. Web Services and the flexibility provided by self-describing entities may be a mean to coordinate an active distributed global schema.

Alternatively, instead of relying on a centralized platform, a more decentralized approach could be adopted. [22] describes a tool for distributed data management and integration. The researchers propose a data model, close to the standard XML model, and a suitable XQuery extension adapted for query processing in a peer-to-peer context. This concept could be extended to include distributed transaction processing. As a guideline, it could be adopted the atomic commit protocol introduced in [46], designed to perform transactions in peer-to-peer environments.

Finally, the primary goal of multi-databases is to provide services to external applications. A different approach would consider defining the requirements of a typical application such as a document collaboration system. Then, design a multi-database system according to those requirements. Consider that, on our knowledge, there is not currently a system allowing authors to modify the same document, the same paragraph, or perhaps the same sentence at the same time. Well suited locking techniques on XML documents would allow a certain degree of granularity such that people could work on the same document. This would be a considerable advance in document collaboration; the flexibility of XML could make this possible, multi-database system would permit authors to collaborate on several distributed databases.

6.2 Conclusions

Evaluation

Although the evaluation could not consider the features provided by an ordinary database, it gives an idea of the performance loss caused by the additional process-

ing in a multi-database system. The global evaluation shows that XML processing and the software logic of platform to carry out a global transaction results in the response time of a transaction to be reasonably greater comparing to the same distributed transaction executed directly on the database.
The scheduler evaluation compares two schedulers in the same case scenario concluding that each is may be more efficient in determinate conditions and in presence of different sets of query/update predicates. Both experiments give indications that an XML-based multi-database system is conceivable.

Self-evaluation

The design section presented a comprehensive software architecture that integrates XML-based technologies to provide a multi-database system. An initial implementation features a transaction processor, a query processor and a connectivity to facilitate the integration of databases that do not support distributed transactions.

Subjectively, the platform implementation is well engineered. In fact, from the software engineering point of view, the software did not turn out to be rigid when major re-factorings were carried out. Utilities developed in support of the whole software were reused in several parts. The logic structure, organized in packages, make the software modular and extensible.

Issues

Most of the issues later translated into time consuming tasks were related to system problems. Apart from installing Oracle, getting familiar with database systems and relative drivers can involve some difficulties and more importantly, it requires time; it is roughly estimated that at least 60% of the time required to implement the platform was spent on infrastructure code.

The fact that a XML native database system supporting node-level locking was not adopted for this project involved many limitations on transaction processing

component. Transactions not causing inter-node conflicts could not be executed concurrently.

When using the XML processing API, one can feel the consistent number of object that have to be created or the number of functions that have to be executed to carry out an apparently simple operations. Using these resources requires the programmer to carefully read the guidelines for optimizing the whole performance of the XML parsing processes. However, often the impression was that the code written using the APIs was not the most effective.

Another purely technical issue was that, as usual, debugging applications in a distributed environment is a tough task. Sometimes, it was difficult to understand what was going on, especially when the DBMSs drivers return exceptions that are difficult to decrypt or when deadlocks occurred.

Design decisions in hindsight

With the benefits of hindsight, considering the effort required to develop the XML-RPC connectivity and the XA support for a database, it would have better to use existing technologies (connectivity and XA support) in order to concentrate on more high-level mechanisms. However, according to a product research, no XML native database supporting XA was found.
This was not really a wrong design decision but more an under-estimation of the time required to implement this platform module.

Goals VS achievements

The goals were partially achieved. A basic implementation of the platform has been implemented raising some challenges related to this research area.

General conclusions

[20] presents a good overview of the database technologies and products currently available on the market. During the choice of the databases for the platform and during the implementation itself, it has emerged the limitation of XML-enabled databases regarding transaction processing. Inappropriate locking mechanisms lead to a poor concurrent access to XML data. In a distributed environment this is not acceptable since a distributed transaction is slowed down if not aborted in case of deadlock by too restrictive locking mechanisms on DBMS. Some XML native databases implementing node-level locking could provide a better transaction throughput and therefore make a XML multi-database more efficient.

A general consideration of the personal experience through this project is the limitation of this immature technology (at least from a XML-enabled DB point of view) that a developer has to face when building a system. While many XML tools are fully available, the current techniques used for XML databases have to improve to compete with traditional database systems.

Bibliography

[1] *Extensible Markup Language (XML).*
http://www.w3.org/XML/.

[2] *W3C - The World Wide Web Consortium.*
http://www.w3.org/.

[3] *Web Services (WS).*
http://www.w3.org/2002/ws/.

[4] *Document Type Definition (DTD).*
http://www.w3.org/TR/REC-xml/.

[5] *XML Schema.*
http://www.w3.org/XML/Schema.

[6] *Tamino XML Server.*
http://www.softwareag.com/

[7] *Document Object Model (DOM).*
http://www.w3.org/DOM/

[8] *Simple API for XML (SAX).*
http://www.saxproject.org/

[9] *The Extensible Stylesheet Language Family (XSL).*
http://www.w3.org/Style/XSL/

[10] *XML Path Language (XPath).*
http://www.w3.org/TR/xpath

[11] *XML Programming Language.*
http://xl.inf.ethz.ch/

[12] *XML-QL: A Query Language for XML.*
http://www.w3.org/TR/1998/NOTE-xml-ql-19980819/

[13] *XQuery 1.0: An XML Query Language.*
http://www.w3.org/TR/xquery/

[14] *XML:DB Initiative for XML Databases.*
http://xmldb-org.sourceforge.net/

[15] *eXist XML DB.*
http://exist.sourceforge.net/

[16] *Apache Xindice.*
http://xml.apache.org/xindice/

[17] *Oracle XML DB.*
http://www.oracle.com/

[18] *dbXML - Native XML Database.*
www.dbxml.com/

[19] *X/Open Distributed Transaction Processing (DTP).*
http://www.opengroup.org/

[20] R. Bourret. *XML and Databases.*
http://www.rpbourret.com/xml/XMLAndDatabases.htm, 2004.

[21] Michael Gertz, Jan-Marco Bremer. *Distributed XML Repositories: Top-down Design and Transparent Query Processing.* Technical Report CSE-2003-20, Department of Computer Science, University of California, Davis, USA.

[22] S. Abiteboul, A. Bonifati, G. Cobena, I. Manolescu, and T. Milo. *Dynamic XML documents with distribution and replication.* ACM SIGMOD Intl. Conference on Management of Data, 527538, ACM Press, 2003.

[23] Torsten Grabs, Klemens Bhm, Hans-Jrg Schek *XMLTM: Efficient Transaction Management for XML Documents.* Proceedings of the 11th International Conference on Information and Knowledge Management, 142-152, 2002.

[24] Sven Helmer, Carl-Christian Kanne, Guido Moerkotte *Evaluating lock-based protocols for cooperation on XML documents.* ACM SIGMOD Record, 58-63, 2004.

[25] Stijn Dekeyser, Jan Hidders *Conflict scheduling of transactions on XML documents.* Proceedings of the 15th Conference on Australasian Database - Volume 27, 93-101, 2004.

[26] M. Nicola and J. John. *XML parsing: a threat to database performance.* IEEE/ACM 12th International Conference on Information and Knowledge Management, 175-178, 2003.

[27] D. K. Fisher, F. Lam, W. M. Shui, R. K. Wong *Efficient ordering for XML data.* Proceedings of the 12th International Conference on Information and Knowledge Management, 2002.

[28] Mark Allman *An evaluation of XML-RPC.* ACM SIGMETRICS Performance Evaluation Review, 2-11, 2003

[29] Jim Gray *Notes on data base operating systems.* In Operating Systems: An Advanced Course, volume 60 of Lecture Notes in Computer Science, 393-481. Springer-Verlag, 1978.

[30] M. Fernandez and J. Simeon and P. Wadler and S. Cluet and A. Deutsch and D. Florescu and A. Levy and D. Maier and J. McHugh and J. Robie and D. Suciu and J. Widom *XML query languages: Experiences and exemplars.* http://www-db.research.belllabs.com/user/simeon/xquery.ps, 1999.

[31] H. Schning. *Tamino - A DBMS designed for XML.* Proceedings of the 17th International Conference on Data Engineering, 149-154, 2001.

[32] H.V. Jagadish et al. *Timber: A Native XML Database.* The International Journal on Very Large Data Bases, vol. 11, no. 4, 274-291, 2002.

[33] M. F. Fernandez, W.-C. Tan, and D. Suciu. *SilkRoute: Trading between Relations and XML.* International World Wide Web Conference, May 2000.

[34] M. Carey, D. Florescu, Z. Ives, Y. Lu, J. Shanmugasundaram, E. Shekita, S. Subramanian. *XPERANTO: Publishing Object-Relational Data as XML.* WebDB (Informal Proceedings), 2000.

[35] J. McHugh and J. Widom. *Query Optimization for XML.* Proceedings of the 25th International Conference on Very Large Data Bases, 315-326, 1999.

[36] F. Frasincar, G-J Houben, C. Pau. *XAL: an algebra for XML query optimization.* Australian Computer Science Communications, Volume 24 ,Issue 2, 49-56, 2002.

[37] T. Bhme and E. Rahm. *Benchmarking XML Database Systems First Experiences.* Proc. 9th Int. Workshop High Performance Transaction Systems (HPTS), http://lips.informatik.uni-leipzig.de:80/pub/2001-31/en, 2001.

[38] H. Lu and al. *What makes the differences: benchmarking XML database implementations.* ACM Transactions on Internet Technology (TOIT), 154-194, 2005.

[39] A. R. Schmidt, F. Waas, M. L. Kersten, D. Florescu, I. Manolescu, M. J. Carey, R. Busse. *The XML Benchmark Project.* Technical Report, http://www.cwi.nl/htbin/ins1/publications, 2001.

[40] *Oracle Database 10g Release 2 XML DB: An Oracle Technical White Paper.* May 2005, http://www.oracle.com/technology/tech/xml/xmldb/index.html.

[41] *Web Services Transactions specifications.* http://www-128.ibm.com/developerworks/library/specification/ws-tx/, 2004.

[42] *A comparison of Web services transaction protocols.* http://www-128.ibm.com/developerworks/webservices/library/ws-comproto/, 2003.

[43] G. Coulouris, J. Dollimore, T. Kindberg *Distributed Systems: Concepts and Design.*, Third Edition, Addison-Wesley.

[44] D. Bell, J. Grimson. *Distributed Database Systems.* 1992, Addison-Wesley.

[45] M. Kifer, A. Bernstein, P. M. Lewis *Database Systems: An Application-Oriented Approach.*, Second Edition, Addison-Wesley.

[46] P. M. Lewis, A. Bernstein, M. Kifer *Databases and Transaction Processig: An Application-Oriented Approach.* 2002, Addison-Wesley.

[47] R. Elmasri, S. Navathe *Fundamentals of Database Systems.*, Fourth Edition, Addison-Wesley.

Appendix A

SimpleXQueryX Example

The XML code below is a transaction definition example using SimpleXQueryX.

```
<?xml version="1.0" encoding="UTF-8"?>
<modifications>
  <update docnamespace="http://oraclexmldb/orcl/product.xml"
   select="/products/product[1]/name">
    <readelement docnamespace="http://sleepycatxmldb/customer/customer.xml"
     select="/customers/customer[1]/orders/item[1]"/>
  </update>
  <update docnamespace="http://oraclexmldb/orcl/productorder.xml"
   select="/orders/order[4]/item">
    <readelement docnamespace="http://leepycatxmldb/customer/customer.xml"
     select"/customers/customer[1]/orders/item[1]"/>
  </update>
  <update docnamespace="http://sleepycatxmldb/customer/customer.xml"
    select="/customers/customer[1]/phone">0791235678</update>
  <readelement docnamespace="http://oraclexmldb/orcl/productorder.xml"
    select="/orders/order[1]/quantity"/>
  <readelement docnamespace="http://oraclexmldb/orcl/productorder.xml"
    select="/orders/order[2]/quantity"/>
  <update docnamespace="http://sleepycatxmldb/customer/customer.xml"
    select="/customers/customer[4]/item[1]">DVD</update>
</modifications>
```

Since the above transaction may not be human-readable, let us consider the equivalent definition.

```
BEGIN TRANSACTION;

write(oracle@/products/product[1]/name,
 read(sleepycat@/customers/customer[1]/orders/item[1]));

write(oracle@/orders/order[4]/item,
 read(sleepycat@/customers/customer[1]/orders/item[1]));

write(sleepycat@/customers/customer[1]/phone, 0791235678);

read(sleepycat@/orders/order[1]/quantity);

read(sleepycat@/orders/order[2]/quantity);

write(sleepycat@/customers/customer[4]/item[1], DVD);

END TRANSACTION;
```

Appendix B

Query Processing

B.1 Oracle Query Translator

The code below reports a fragment of the XSL script charged to convert the query definition used by the platform into statements that can be executed on Oracle XML DB. In this case, the query script process a 'update' predicate. After the XSL code, it is reported the update predicate used by the platform (SimpleXQueryX) and the result after the transformation into an Oracle update predicate.

XSL fragment

```
<xsl:template match="update">
  <xsl:variable name="table">
    <xsl:call-template name="namespaceparser">
      <xsl:with-param name="docnamespace">
        <xsl:value-of select="@docnamespace"/>
      </xsl:with-param>
    </xsl:call-template>
  </xsl:variable>
  <xsl:value-of select="concat('update ',
    $table,
    ' set object_value = updateXML(object_value,',
    $apos,@select,'/text()',$apos,',',$apos,.,$apos,')')"/>
```

```
</xsl:template>
```

Update predicate defined through the SimpleXQueryX format (before XSL transform)

```
<modification>
  <update
        docnamespace="http://kdeg.cs.tcd.ie/oraclexmldb/orcl/product.xml"
        select="/products/product[1]/name">product4</update>
</modification>
```

Update predicate ready to be executed on Oracle (after XSL transform)

```
update product set object_value =
        updateXML(object_value,
        '/products/product[1]/name/text()',
        'product4')
```

B.2 SleepyCat Query Translator

The Jave code below shows how the SleepyCat translator operates al the lower level. A generic class (*SimpleXQueryXParser*) parses a statement defined in XML with SAX and it calls the abstract method *xUpdateUpdate* whenever there is a match.

The class *SleepyCatSimpleXQueryXProcessor* specifically implemented for SleepyCat databases inherits from the superclass and overwrite the methods including the method *xUpdateUpdate* as shown in the example. Note that the abstract class *SimpleXQueryXParser* is conceived to be re-usable for other database drivers.

```
public boolean xUpdateUpdate(String xmlDoc, String xPath, String textNode) {
  boolean executionOutcome = false;
  try {
    XmlModify mod = ds.xmlManager.createModify();
    XmlQueryExpression select =
        ds.xmlManager.prepare(xmlTransaction, xPath, qc);
    mod.addUpdateStep(select, textNode);
    XmlDocument retDoc =
      ds.xmlContainer.getDocument(xmlTransaction, xmlDoc);
    XmlValue docValue = new XmlValue(retDoc);
    mod.execute(xmlTransaction, docValue, qc, uc);
    executionOutcome = true;
    } catch(XmlException e) {
      ds.getLogger().error("xUpdateUpdate: execution failed", e);
    }
    return executionOutcome;
}
```

Appendix C

Transaction scheduler

Unlike the simple scheduler, the scheduler executing query/update predicates concurrently uses a slightly complex group of objects. It is therefore provided an overview of how it operates and the objects it uses. Figure C.1 shows its class diagram.

The main class, *ParallelScheduler* (implements the interface *XMLDBScheduler*), receives the transaction that has to be executed. This class contains four 'private' functions called in the same order as they are listed in the class diagram.
openAndExecute opens the resource managers and starts the transaction branches according to the statements it receives. Additionally, this function is charged to create the schedule and to eventually execute the queries according to that schedule. Note that *abortGlobalTransaction* can be called at any stage in case of failure. The functions *endAndPrepare* and *commitAll* do not deal with the schedule but they just follow the two-phase commit directives (see the algorithm in section 4.4.2).

As mentioned, the function *openAndExecute*, among other things, prepares the schedule. Figure C.1 includes the classes needed to organize statements in steps that will be executed sequentially. A step may contain as many threads as the number of databases that are involved in the distributed transaction (*ParallelStatementExecutor*). Thus, each thread will query/update the corresponding database. An example

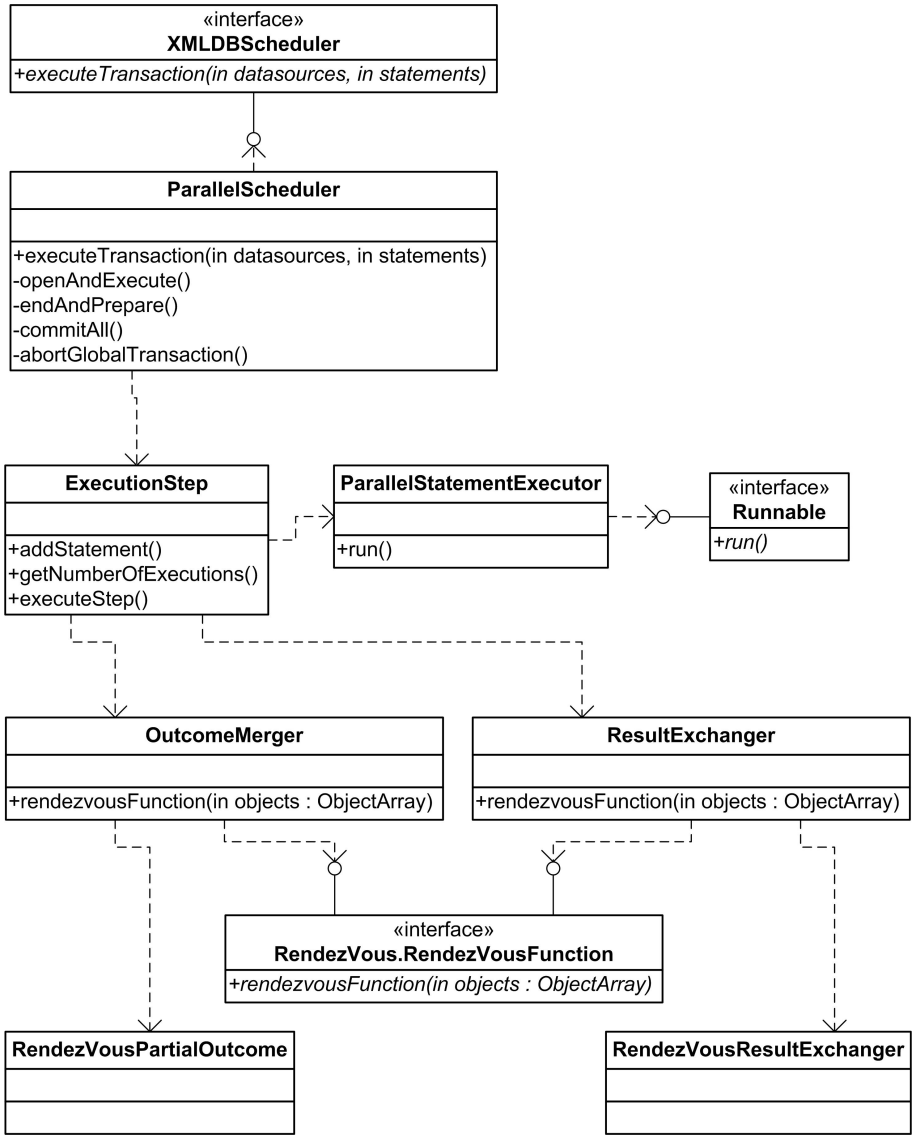

Figure C.1: Scheduler class diagram.

of a concurrent schedule is shown in Figure 3.9.
openAndExecute assigns every statement to an execution step. The basic principle for the creation of steps is that if a given statement depends on another statement that has to be executed on a different database, then a new step is needed. This because two operations may have to synchronize (e.g. exchange a partial value or variable). In contrast, if statements are independent, they can belong to the same execution step and be eventually executed on each database in a concurrent manner.

There are two kinds of synchronisations. The simplest one consists in assessing whether or not a partial failure has occurred (*OutcomeMerger* and *RendezVous-PartialOutcome*). The more complex synchronisation point is designed to exchange partial results between threads (*ResultExchnger* and *RendezVousResultExchanger*). Both synchronizations are implemented with a so called rendezvous or synchronization barrier (see section 4.3 for the library that has been used). Basically, a rendezvous is arranged between a defined numbers of threads. When they ALL reach the synchronization point, the rendezvous takes place. At this stage, the threads can finally communicate together (the function of the object implementing the interface *RendezVous.RendezVousFunction* is executed in order to exchange objects). If one or more thread(s), for some reasons (e.g. distributed deadlock), can not attend the rendezvous, the other participants wait at the synchronization barrier for a certain amount of time before giving up and generating an exception. In this case, there is a failure and the global transaction is aborted. If all steps are executed successfully, the whole transaction can be prepared and committed (if any failure occurs during the rest of the two-phase commit protocol).

Appendix D

XML Documents

This appendix lists the XML documents that have been used for the evaluation of the system.

On Oracle: product.xml

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<products>
    <product id="1">
        <name>product1</name>
        <price>1435</price>
    </product>
    <product id="2">
        <name>product2</name>
        <price>5412</price>
    </product>
    <product id="3">
        <name>product3</name>
        <price>8923</price>
    </product>
    <product id="4">
        <name>product4</name>
        <price>2657</price>
    </product>
```

```
</products>
```

On Oracle: productorder.xml

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<orders>
    <order id="1">
        <item>product4</item>
        <quantity>2000</quantity>
        <date>12.12.2005</date>
    </order>
    <order id="2">
        <item>product3</item>
        <quantity>134</quantity>
        <date>12.10.2005</date>
    </order>
    <order id="3">
        <item>product2</item>
        <quantity>435</quantity>
        <date>01.01.2006</date>
    </order>
    <order id="4">
        <item>product1</item>
        <quantity>780</quantity>
        <date>23.11.2005</date>
    </order>
</orders>
```

On SleepyCat: customer.xml

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<customers>
    <customer id="6">
        <name>John</name>
```

```
        <phone>014567890</phone>
        <orders>
                <item id="1">product4</item>
                <item id="2">product3</item>
        </orders>
    </customer>
    <customer id="8">
        <name>Jenny</name>
        <phone>017561290</phone>
        <orders>
                <item id="3">product2</item>
        </orders>
    </customer>
    <customer id="90">
        <name>Joe</name>
        <phone>014236576</phone>
    </customer>
    <customer id="56">
        <name>Paul</name>
        <phone>011287452</phone>
        <orders>
                <item id="1">product4</item>
                <item id="4">product1</item>
        </orders>
    </customer>
</customers>
```