Value-oriented XML Store

Kasper Bøgebjerg Pedersen
IT-University of Copenhagen

Jesper Tejlgaard Pedersen, c960709
The Technical University of Denmark

Supervisors:
Michael R. Hansen
Fritz Henglein

November 2002

Abstract

This thesis presents a distributed value-oriented storage facility for storing XML documents called XML Store. XML documents stored in XML Store can be accessed and manipulated using an API, called the Document Value Model (DVM). Value oriented programming is programming with immutable values and references to values.

This thesis identifies key concepts in designing an implementing such a storage facility.

The XML Store prototype presented in this thesis is evaluated by building a sample application and performance testing.

Evaluation showed that the value-oriented programming model have advantages over the traditional imperative model, when programming distributed XML applications.

Acknowledgements

A few people have influenced this thesis and we are grateful for their contributions.

The guidance and advice (technical and non-technical) of our respective supervisors Michael R. Hansen and Fritz Henglein have been skill full and very important to the outcome of the thesis. Fritz Henglein further provided initial inspiration for the thesis, which we are grateful for.

During the period of the thesis much time have been spent with Mikkel Fennestad, Tine Thorn and Anders Baumann. We thank them for their inspiration and fruitful discussions of value-oriented programming, manipulation and storage of Extensible Markup Language (XML) documents and XML related technologies.

We would like to thank Brian Christensen, Thejs W. Jansen, Petar Kadijevic and Kinnie Bak Pedersen for reviewing different versions of the thesis and giving valuable feedback.

Preface

This Master Thesis concerns the development and implementation of a value-oriented storage facility for storing XML documents in distributed environments.

The Master Thesis is the outcome of a project across universities. It has been written by Jesper Tejlgaard Pedersen from the Technical University of Denmark (DTU) and Kasper Bøgebjerg Pedersen from the IT-University of Copenhagen (IT-C).

Michael R. Hansen from the Institute of Informatics and Mathematical Modeling (IMM) at the Technical University of Denmark and Fritz Henglein from the Department of Computer Science at the University of Copenhagen (DIKU) has been supervising the thesis. Michael R. Hansen has been the primary supervisor of Jesper Tejlgaard Pedersen, while Fritz Henglein has been both the primary (and only) supervisor of Kasper Bøgebjerg Pedersen and secondary supervisor of Jesper Tejlgaard Pedersen.

The thesis started February 1st 2002. It has been submitted by Jesper Tejlgaard Pedersen August 31st 2002. Kasper Bøgebjerg Pedersen submitted another version of the thesis November 1st 2002. The main differences are improved performance of the prototype implemention (these results are presented in chapter 8) and general improvements of the report.

Readers of this thesis are assumed to have basic knowledge of object oriented programming, distributed programming, operating systems and elementary computer architecture. Examples through out the thesis (and the source code) are written in the Java programming language. Besides Java, basic knowledge of functional programming languages such as ML may provide a small advantage.

The thesis and source code are available at:

`http://www.it-c.dk/people/kasperp/xmlstore`

Contents

Chapter 1

Introduction

XML

The *Extensible Markup Language* (XML)[1] was designed by the W3 consortium as a universal format for structuring and exchanging data on the Web. The need for a standard became evident as the amount of data with different formats exchanged on the web grew.

Data may differ in a number of ways: Data has a number of different forms, everything from unstructured data stored in native file systems to highly structured data stored in relational databases. Furthermore many applications store their data in some proprietary form (an obvious example is documents stored with Microsoft Word). This poses problems when exchanging data.

Most data available on the web is semi-structured and as such does not fit the strict data model of relational databases.

XML provides a standard format for representing semistructured data in a platform-independent fashion. XML consists of two components:

1. A model for representing tree-structured data (*XML Information Set*[2]).
2. A linear syntax for representing the model.

Programming with XML documents

Several technologies for manipulating XML documents exists, the *Document Object Model* (DOM)[3] and *Simple API for XML* (SAX)[4]. DOM is the official proposal form W3C and treat documents as updateable objects. SAX is, as revealed by its name, more simple; it/this technology provides an event driven model for accessing XML documents. SAX is basically a lexer for XML documents. DOM is an interface specification and current implementations let application programmers manipulate an in-memory representation of XML documents and is often used in combination with an XML parser.

No standard for persisting XML documents exists. Documents are usually stored as flat files in the native file system or mapped to relational databases.

These issues enforces most XML applications to follow a certain process:

1. Read XML data from some external source, file system, network etc.

2. Parse data into an internal representation. In DOM, an in-memory abstract syntax tree representing the XML Information Set.

3. Traverse and manipulate this representation. Manipulation may construct new in-memory representations.

4. Unparse and write data to some external source

Chapter 2 explores shortcomings of current APIs for working with XML.

Distributed systems

Today's world of computation is distributed. Systems in which multiple machines share common resources are widespread. The success of the Internet has bred a need for almost all applications to communicate with the outside world.

Traditionally distributed systems have a *client/server* architecture[5], where a central *server* offers a service to a number of *clients*. The server has to handle many (simultaneous) client requests, therefore a client/server architecture requires servers to be powerful. Furthermore, servers are "single points of failures", i.e. if the server fails the systems cannot be used. An example of client/server systems are web servers that handle requests from a (potentially) large number of clients (web browsers).

In recent years distributed systems with *peer-to-peer* architecture have risen in popularity, due to the popularity of file swapping systems such as Napster[6], Gnutella[7] and Freenet[8]). The term *peer*, to have equal status, is used to characterize participants in a decentralized distributed system, in which these participants have equal status and all communication is symmetric. Such a system have a peer-to-peer architecture. For such a system to function, each peer must act as both client and server.

Common to peer-to-peer systems is that they

- have no centralized server architecture[1]
- address isues such as active replication and opportunistic replication (caching) to achieve fault tolerance, high availability, performance and resilience against attacks[9].
- rely on cryptographic techniques to ensure authentication and enforce access controls.

Peer-to-peer systems are more fault tolerant than client/server systems as they eliminate the "single point of failure". Furthermore, a large number of peers can crash or leave the system without destroying the whole system, this (potentially) gives peer-to-peer systems a higher degree of availability.

Replication of data is central to distributed systems (especially in peer-to-peer systems), in order to provide performance and fault tolerance.

Most distributed systems adopt an imperative programming model, one that revolves around destructive updates of variables. In such systems keeping replicated data consistent becomes complex, especially under transaction control. Keeping data consistent is a general concern in systems where data are shared. In such system complex transaction mechanisms are often introduced to help keep data valid.

[1] This does not apply to Napster.

Value-oriented programming

Value-oriented programming is programming with values and references to such, as known from functional programming. A *value* describes an entity that cannot be updated, instead updates have to be coded as construction of a new value.

Problems with validity of data in distributed system can be traced back to the destructive updates of the imperative programming model. In a value-oriented programming model, one that considers values as immutable entities, issues of validity become simple, since values are not updated. Updates are performed by updating a/the? variable, which holds a reference to a value, with a reference to another value. This can be done atomically.

Value-oriented programming has advantages over the traditional imperative model, when building distributed systems;

- Light weight replication and caching of values, as no coherence protocols are needed.
- Validity of data, as data cannot be destructively updated, it cannot become invalid. Updating a variable that holds a reference to a value can be done atomically.

Distributed Value-oriented XML Store

XML documents (which are trees), can be treaded in a value-oriented fashion, in contrast to the Document Object Model (DOM). Considering XML trees in a value-oriented fashion means, considering the root of trees and all subtrees as values. The root of a tree has references to its subtrees. References to values can be made persistent and stored on disk. This allows:

- Storing XML documents "natively", such that the tree structure is kept and can be traversed on-disk (as opposed to only "left-to-right" preorder traversal used when serializing XML documents).
- No parsing of documents from a serialized representation to an internal representation is needed before processing.
- Sharing of XML tree nodes within and across documents.

The ability to traverse XML documents on-disk eliminates the need to keep whole XML documents in-memory while processing.

The idea of a distributed value-oriented XML Store, is to build a storage manager, that provides transparent distribution and persistence, for XML documents.

1.1 Problem statement

The main goal of this thesis is to prove that a value-oriented programming model has several advantages over the more traditional imperative model when working with XML data in a distributed setting. We will provide a value-oriented Application Programming Interface (API) for manipulating XML data, called *Document Value Model* (DVM), and show how this API solves disadvantages

of current DOM implementations. We will prove this by building a distributed value-oriented storage manager for XML documents, called XML Store, and test it against conventional ways of working with XML documents.

The implementation of XML Store provided with this thesis is a prototype, where emphasis has been put on identifying key concepts and designing a flexible and extensible system.

We intend to evaluate the following:

1. Usability and adequacy of DVM in contrast to the Document Object Model (DOM) and Simplified API for XML (SAX). We will illustrate that even simple applications require complex solutions using DOM and SAX, and illustrate how DVM gives more elegant and efficient code. This will be evaluated through implementation of a running example - a dictionary application that offers keyword search and functionality to insert new words. We will refer to the example throughout the thesis. Other sample applications will be implemented to evaluate the API.

2. Evaluate efficiency of the storage strategy, by conducting tests, which store and access different documents.

1.1.1 XML Store desiderata

This section lists desired properties of XML Store.

Decentralized The XML Store network should be a fully decentralized peer-to-peer network.

Distributed persistence The XML Store must follow general requirements for distributed file systems [5, p. 315-316]: transparency, scalability, efficiency, replication, consistency, security and fault tolerance.

Efficient and transparent sharing of XML documents Sharing parts of documents both within documents and across documents should be efficient and transparent.

Convenient and adequate API The API provided by the XML Store must offer operations that are convenient and adequate when working with XML-data. The API must be value-oriented.

Hide location and distribution Documents should, from an application programmer perspective, be treated equally no matter their location, be that in-memory, on-disk or across a network.

Lazy loading. XML documents must be loaded lazily (on request), to prevent whole documents from being loaded into main memory.

No parsing and unparsing of XML documents. XML documents should be stored natively in the XML Store to prevent excessive parsing and unparsing.

Configurable. The XML Store architecture must be configurable and extensible, in such a way that functionality such as caching and buffering can be combined differently on each XML Store.

1.1.2 Limitation

A number of aspects of XML Store are not considered in this thesis. Mobility of data, distributed garbage collection, routing algorithms, security threads and facilities for querying document stored in XML Store are not addressed in this thesis.

A range of topics are described in the thesis, but have not been implemented. These are: implementing an optimistic locking functionality in the name server, read and write buffering, asynchronous writes to disk and inlining of child nodes when saving XML documents.

Some modules or single entities have not been implemented optimally. These include the name server and the representation of child nodes. The problems with the implementations are analyzed and better solutions are proposed.

Focus in this report is on identifying key concepts of XML Store and an API to access XML documents. During development of the XML Store prototype we have used unit testing to ensure correctness. We have used frameworks to automate the test and build process. These unit tests have not been documented as testing (no matter how important) is beyond the scope of this report. The prototype implementation should be seen exactly as such, it can be used to illustrate concepts discussed in this thesis, not as a finished product.

1.2 Flavor of the Document Value Model

This section gives a quick sample of the flavor of the Document Value Model (DVM) provided with this thesis, through a simple example.

The example creates and prints the following XML document, using DVM.

```
<greeting>Hello World!</greeting>
```

Creation of Nodes is done using a factory[10, p.87].

```
XMLStoreFactory factory = XMLStoreFactory.getInstance();
Node hello = factory.createCharDataNode("Hello World!");
Node greeting = factory.createElementNode("greeting", hello);
```

Traversing and printing the document to standard out is equally simple, retrieve the element name (with `getNodeValue()`) then retrieve the value of its first child, and print.

```
String tagName = greeting.getNodeValue();
System.out.println("<" + tagName + ">" +
                    greeting.getChildNodes().getNode(0).getNodeValue() +
                   "</" + tagName + ">";
```

Changing the greeting to say “Extensible Markup Language”, must be coded by creating a new node, instead of destructively updating the original, as DVM is value-oriented.

```
Node xml = factory.createCharDataNode("Extensible Markup Language");
greeting = factory.createElementNode(tagName, xml);
```

DVM offers utilities to do this more elegantly, however the basic principle is the same. The full document value model is described in chapter 4.

1.3 Report guide

This section serves as a brief guide of the report. The report has the following chapters:

Chapter 2 - Extensible Markup Language provides background information on the Extensible Markup Language (XML), presents different approaches for working with XML and states shortcomings of these.

Chapter 3 - Value-oriented programming presents and defines the value-oriented programming model, specifies how to work with trees using the model. Describes value-oriented programming in a distributed environment.

Chapter 4 - The Document Value Model presents a value-oriented API for manipulating XML documents in a value-oriented fashion. The chapter serves as a Document Value Model user guide.

Chapter 5 - XML Store architecture discusses design of a distributed XML Store. Focus is on the design decisions made to build the prototype implementation given with this thesis.

Chapter 6 - XML Store implementation describes implementation details of the XML Store prototype.

Chapter 7 - Evaluation of the Document Value Model evaluates DVM by building sample applications.

Chapter 8 - Experimental results tests the implemented prototype of XML Store.

Chapter 9 - Future works presents issues for future research.

1.4 Related work

XML Technologies

As mentioned above, different technologies for manipulating XML data exists.

Document Object Model (DOM). The official specification from W3C for an Application Programming Interface (API) for manipulating and accessing XML documents. DOM offers an imperative tree-oriented object abstraction of XML documents. DOM is a platform and language independent specification. Most DOM implementations build an in-memory abstract syntax tree representing the XML Information Set, which can then be accessed and manipulated.

DOM is a complex specification. It consists of three different levels (and an unofficial level 0) and each level extends the specification from the previous level. Level 0 is a vague API for accessing certain elements of an HTML document (ECMAScript), Level 1 specifies a fundamental object model for a document. Level 2 extends core functionality of level 1, and adds Views, Events, Style, Traversal and Range. Level 3 adds access

to entities, DTDs and Schemes, XPath and load and save of documents [11, 12]. According to W3C[13] DOM identifies:

- the interfaces and objects used to represent and manipulate documents.
- the semantics of these interfaces and objects - including both behavior and attributes.
- the relationships and collaborations among these interfaces and objects.

Simple API for XML (SAX). An original Java-only API which is now a de facto standard[4]. SAX is a low-level event based API, and is mainly used to process large XML documents, as it does not build any internal representation of the document processed.

XSL Transformation (XSLT)[15]. A style sheet language for defining transformations from one class of XML documents into another class of documents [11]. XSLT is a declarative language, i.e. you state what you want not how you want it done.

XSLT can be implemented using DOM and SAX, as in Xalan-Java[29].

XQuery[16] is the official proposal for an XML query language. The ability to query XML data becomes important as the amounts of XML documents stored and exchanged increases. XQuery is

> *designed to be a small, easily implementable language in which queries are concise and easily understood. It is also flexible enough to query a broad spectrum of XML information sources, including both databases and documents*[16].

Storing XML

When using XML in real world applications, it is necessary to enable data to "survive" the life of a single process and to allow data to be shared between multiple processes. Therefore it is important to be able to persist XML data. A number of different methods for persisting XML data exists, these are discussed further in chapter 2

The simplest of these keep serialized versions of XML documents in a flat files. This requires excessive need for serializing/unserializing documents, when data is being manipulated.

Another method used is to keep XML data in relational databases, which may require complex mappings from XML *entities* to database *tables*.

Currently many efforts focus at development of alternative XML storage managers, most of these efforts revolves around *Native XML Databases (NXD)* which are databases designed especially to store XML documents[17]. Native XML Databases support XML query languages, e.g. XQuery

Peer-to-peer file systems

In recent years peer-to-peer file systems have risen in popularity, due to the success of file-sharing systems such as Napster[6], Gnutella[7] and Freenet[8].

The first widely spread peer-to-peer file sharing system, Napster, actually uses a central server to store an index of available files in the systems. This index stores filename and location (IP of machine) of the file. Users search this index for filenames and obtains a location of the file, the file can then be retrieved from the machine holding the file. As the process of locating files is centralized, Napster cannot be considered a "pure" peer-to-peer system.

Gnutella has no centralized server storing locations of files. It is a completely decentralized and peers rely on broadcast communication, when locating files.

Other peer-to-peer systems such as Freenet, PAST[18], CFS[19] and the XML Store implementation given by Baumann et. al.[9] provides location transparent storage of data, with efficient location independent *routing* in a decentralized environment.

1.5 Conclusion

This thesis presents a distributed value-oriented storage facility, called XML Store. XML Store can be used to persist and access XML documents. XML Store offer a value-oriented API, called Document Value Model (DVM), to access and manipulate stored XML documents.

The main focus of this thesis has been to examine possible advantages of value-oriented programming against traditional imperative programming when building distributed XML applications.

Value-oriented programming is programming with values and references to values. Values are immutable entities. In value-oriented programming updates must be coded as creation of new values, in contrast to imperative destructive updates.

An XML Store prototype has been buildt, as a peer-2-peer network using IP Multicast to communicate between peers.

XML Store has been evaluated through implementation of simple applications and has been compared with similar applications implemented with DOM. These illustrated advantages of the value-oriented programming:

Memory XML Store applications can handle XML documents of arbitrary size, without extra effort of application programmers.

Sharing Nodes (sub trees) may be shared between and within XML Documents stored in XML Store, as opposed to DOM.

Transparency XML Store offers location and distribution transparency. I.e. where documents are located (locally/distributed or on disk/in memory) is transparent to applications programmers. An application buildt for a local environments works in distributed environment without any changes.

Concurrency control As values cannot be updated, application programmers does not have to worry about concurrency.

The evaluation also showed that the name server introduces a concurrency problem, we have proposed an improved name server that solves this problem.

We expected other advantages of value-oriented programming, specifically we expected that caching could be implemented without concern to coherence as values are immutable. To evaluate this we ran a number of experimental performance tests. These tests illustrated advantages of the value-oriented programming model. Modifying persisted documents and storing the result is fast. This is an effect of sharing. The tests also showed that repetitive access of the same value is fast as the values is cached. The tests also illustrated high loading times for distributed documents, future work is required to improve the distributed load performance.

We have shown that a distributed value-oriented storage facility can be constructed. We have shown value-oriented programming may ease development of distributed XML applications compared to the imperative programming model. The value-oriented programming model allows simple replication and caching strategies, sharing of values, removes the need for concurrency control by providing atomic updates and provides transparent distribution of values.

Chapter 2

Extensible Markup Language

Extensible Markup Language (XML)[1] is a platform-independent language for describing semistructured data such as documents (books), messages for interchange between different computers etc. XML is a *markup* language derived from *Standardized General Markup Language* (SGML)[20], this means, it uses tags to enclose text in a document. XML allows the definition of sub languages through a number of technologies and is thus often also called a metalanguage, that is, a language for defining languages. XML has a large number of usage patterns:

1. Represent documents (that is conceptual "real" documents), such as books, orders, customers etc.
2. Represent programming languages and protocols, (e.g *Extensible Stylesheet Language (XSL)*[21], *Simple Object Access Protocol (SOAP)* [22]).
3. Represent the layout of documents, e.g. *The Extensible HyperText Markup Language (XHTML)*[23], *Scalable Vector Graphics (SVG)* [24].

XML has found widespread acceptance with support from basically all major software, operating system and database vendors. With increasing demand for distributed systems it establishes a common, platform-independent way of exchanging data, including messages, between heterogeneous and loosely coupled systems.

This presentation considers *Minimal XML* [25], which is a strict subset of XML. Minimal XML is defined by the grammar in figure 2.1.

The basic construct of an XML document is the *element*. An XML document contains one element, often called *root element*. An element is enclosed in *tags*, defining the element's name. The start tag consists of the element name enclosed in `< >`. An element with name `keyword` thus has the start tag `<keyword>`. End tags are prefixed with `</` and their names must match the corresponding start tag. The term *content* is used to describe the inside of an element (everything between the start and the end tag). Content may be a sequence of elements with optional white spaces between elements, or a sequence of *character data* and *character references*. Elements that are content of other elements are called

```
document ::= WS* element WS*
element ::= STag content ETag
STag ::= '<' Name '>'
ETag ::= '</' Name '>'
content ::= (element | WS)* | (CharData | CharRef)*
Name ::= [^<>&/]+
CharData ::= [^<>&]
CharRef ::= '&#' [0-9]+ ';'
WS ::= (#32 | #9 | #13 | #10)
```

Figure 2.1: Minimal XML grammar[25]

sub elements, *child elements* or *nested elements*. Character data is plain text not including `'<' '>'` and `'&'`. Character references are references to characters. Character references may be used to refer to characters which cannot otherwise be written e.g. `'&#60;'` represents the reserved character `'<'`.

Minimal XML is as mentioned a subset of XML and does not include *Attributes*, *CDATA Sections*, *Comments*, *Document Type Declarations*, *Empty-Element Tags*, *Entity References*, *Mixed Contents*, *Predefined Entities*, *Processing Instructions*, *Prolog* and *XML Declaration*[25]. For a full definition of XML, see the XML specification[1].

```
<dictionary>
  ...
  <word>
    <keyword>foo</keyword>
    <desc>
       <p>
         <type>jargon</type>
         /foo/ A sample name for absolutely anything,
         especially programs and files ...
       </p>
       ...
    </desc>
  </word>
  ...
</dictionary>
```

Example 2.1: Sample XML document, root node dictionary. ("..." symbolizes more text or more nodes

The XML supported by the prototype implementation supplied with this thesis is Minimal XML plus attributes and mixed contents.

Attributes are name to value bindings. Each element may hold a number of attributes. Attributes differ from sub elements in a number of ways. Attributes are unique, i.e. the same attribute can only occur once in the same element, attributes cannot be nested and they can only hold character data.

Mixed contents allow elements to contain character data, optionally interspersed with child elements. [1]

Consider example 2.1 which shows XML representing the word "foo" in a dictionary [26] (Appendix B defines and describes the dictionary). The example illustrates an XML document that has six elements `dictionary`, `word`, `keyword`, `desc`, `p`, and `type`. Elements `keyword` and `desc` are sub elements of `word`.

Example 2.1 does not contain any attributes, attributes are expressed like `<word keyword="foo"> .. </word>`.

What is normally referred to as XML documents has two forms, a serialized representation of some data, as described above, and an abstract model of the document. The abstract model is a model of labeled trees, and is defined as *XML Information Set*[2], section 2.1 describes this model of XML documents.

Writing programs that use XML documents as a data source is done using technologies developed for this purpose, section 2.2 describes and analyzes such technologies. Considering XML documents as data sources for computer programs, documents may be stored on some external device. Section 2.3 discuss different approaches used to store XML documents.

2.1 XML documents as trees

As mentioned above, XML is a language for describing semistructured data. In this context XML documents can be regarded as an abstract data type. The *XML Information Set* specification [2] does this as it

> *... defines an abstract data set called the XML Information Set (Infoset). Its purpose is to provide a consistent set of definitions for use in other specifications that need to refer to the information in a well-formed XML document.*[2]

An XML-document is well-formed when

1. It only contains one top-level element, this element name must be unique (often called root element)
2. Tags are properly balanced
3. Attribute names are unique and their values quoted

Basically, the requirement for well-formedness ensures that serialized XML documents can be transformed into labeled trees [27, p.29].

Figure 2.2 illustrates the serialized XML document from example 2.1 as a labeled tree.

The term "XML documents" commonly refers to two components, either a model of tree structured data or a serialized representation of this model. In this report we consider XML documents to be models of labeled ordered trees (directed acyclic graphs (dags) when sharing of subtrees is considered).

2.2 Programming with XML

One of the reasons for the popularity of XML is the suite of related standardized technologies used when programming with XML. Programming with XML is writing programs that consider XML documents as a data source, i.e. programs

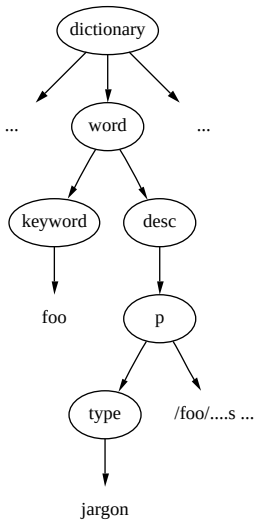


Figure 2.2: Labeled tree representation of XML document. Root element `dictionary` has child elements `keyword` and `desc` ("..." symbolizes more nodes)

that access and manipulate XML documents. Programming with XML is done using related technologies. The two main Application Programming Interfaces (APIs) used to access XML documents are the *Document Object Model* (DOM) [3] and *Simple API for XML* (SAX) [4].

This section will present these two distinctively different ways of working with XML, and evaluate both. The APIs are evaluated by implementing a simple dictionary that supports search for keywords and inserts of new words. Description of dictionary, application and full source code resides in appendix B. Besides DOM and SAX, we also consider the *Extensible Stylesheet Language (XSL)* which is a programming language designed to transform one XML document into other documents (that may be other XML documents).

DOM is described in section 2.2.1, SAX in section 2.2.2 and XSL in section 2.2.3.

2.2.1 Document Object Model

The Document Object Model (DOM) is a high level tree-oriented interface for accessing and updating content, structure and style of documents. DOM is

> *[..] a platform- and language-neutral interface that will allow programs and scripts to dynamically access and update the content, structure and style of documents.*[3]

This means that DOM is a language independent specification for manipulating XML documents.

DOM provides, as the name reveals, an object model of an XML document. This object model introduces meaningful abstractions of XML constructs, such as elements, attributes, text etc. and specifies operations for accessing and

modifying the XML document. DOM is a large specification, providing access to all parts of XML documents. This description of DOM is an overview of programming with DOM and hence only covers the basics. For full reference see the DOM specification [3].

To use the DOM interface, a serialized XML document must be parsed into an internal representation of the document, referred to as the *DOM tree*. A DOM tree is a representation the XML infoset of a given document. Although not part of the specification, DOM implementations[1] require that the entire XML document is parsed into memory, before the document may be processed.

The API consist of a number of interfaces, each representing part of an XML document. The `Node` interface is central. Elements, text, attributes etc. all extend the `Node` interface. The DOM interface also contains additional interfaces for each node type (Element nodes, text nodes, etc.). These interfaces provide more convenient operations for accessing and manipulating documents. In the Java implementation getting access to these sub operations requires an explicit downcast (from `Node`), since they are subtype of `Node`.

To illustrate and evaluate DOM a sample application is implemented. This application is, as mentioned above, a dictionary that supports (keyword) lookup and inserts of new words (see appendix B for description and full source code). The XML document representing the dictionary is structured as the example shown in example 2.1. When implementing keyword search, the following steps are carried out:

1. Parse the entire database (7.6 Mb XML document) and build a DOM tree.
2. Perform binary search on keywords (as these are sorted).
3. Build and return result (result may be "no match found").

These steps are illustrated by code in example 2.2. Lines 2-6 parse and builds an in-memory DOM tree. Line 8 performs binary search, which returns `null` on unsuccessful searches. The binary search method is not shown here. The rest of the code builds a result. On successful search the found word node must be cloned (line 12, `importNode` clones) as DOM does not allow sharing of nodes. The node is appended to a `Document` node and returned (line 12-13). In case of unsuccessful searches a DOM tree corresponding to `"<word><keyword>No match found</keyword></word>"` is created and returned (lines 16-21).

Parsing a 7.6 MB document each time a search is performed takes time. Several approaches can be used to speed up the search process (but they all have to be implemented by application programmers). The most simple way to speed up searches is to keep the entire DOM tree in memory, and only parse it once (e.g. when the application starts). This would work, if the dictionary is never updated or if not shared between several processes as in a distributed environment. Even so keeping a large dictionary in memory may not be feasible. Another way to speed up searches would be to keep smaller files, (e.g a file for each letter in the alphabet). These could then work as indexes, as the search function then only has to parse the correct (and much smaller) file. The solution requires work from the application programmer as code for choosing the correct file must be implemented. This solution is only temporary, when new words

[1] during this report we have tested the DOM implementation in J2SDK1.4 [28] package `org.w3c.dom`

```
1  public Document keywordSearch( String keyword ){
2    DocumentBuilderFactory factory =
3        DocumentBuilderFactory.newInstance();
4    factory.setIgnoringElementContentWhitespace(true);
5    builder = factory.newDocumentBuilder();
6    Document dict = builder.parse( new File( dictName ) ) ;
7
8    Node match = binarylookup( dict, keyword );
9
10   Document doc = builder.newDocument();
11   if( match != null ){
12     Node n = doc.importNode( match, true );
13     doc.appendChild(n);
14     return doc;
15   }
16   Node word = doc.createElement("word");
17   Node keyword = doc.createElement("keyword");
18   keyword.appendChild(doc.createTextNode("No match found");
19   word.appendChild(keyword);
20   doc.appendChild(word);
21   return doc;
22  }
```

Example 2.2: Code to find keyword in dictionary. Code is simplified for illustration purposes, e.g. is free of imports and Exceptions

are inserted, the size of each index file grows and parse time increases. A clever scheme for splitting index files when these grow to a certain size could be implemented.

Inserts follow much the same path as search, that is

1. Parse and build the DOM tree,
2. Find the place to insert by performing a binary search of the keyword to be inserted.
3. Insert the word, by building a DOM tree of the word to be inserted, and insert it into the existing dictionary tree.
4. Unparse and write the dictionary (7.6 MB and growing) to disk.

No concurrency control on insert is implemented. Such must also be implemented by the application programmer. This makes it difficult to build even small applications (e.g. building this insert function for web use) using DOM.

To speed up the searches and inserts and to help with concurrency issues, XML documents are often stored in a relational database system because those provide persistence management (automatic loading and saving from disk, driven by data actually used) and concurrency control (transaction management) (see section 2.3.2).

The above introduction and evaluation of working with DOM (using the current implementation) lead to the following disadvantages of programming with DOM:

- serial access to persisted XML documents only ("from left to right") due to XML documents being stored in a serial fashion, unless indexes are implemented by application programmers.

- severe main memory requirements since whole documents has to be read into main memory before processing (including pieces never used).
- frequent serialization and deserialization processing of XML documents (changes between serial and tree-oriented representation).
- no sharing of XML documents or parts of XML documents, leading to expensive copying of parts of XML documents and the need for repetitive (cumbersome to program and inefficient) update processing of cloned nodes.
- fragile code due to the need for updating DOM-trees and carefully synchronize/coordinate the order and nature of updates.

The main advantage of DOM is that the API is fairly straightforward and natural to use given an understanding of object oriented programming. The abstractions of nodes is natural and works well.

2.2.2 Simple API for XML

Shortcomings of DOM, particularly problems with processing arbitrarily large XML documents, motivate application programmers to use other technologies. One of these is Simple API for XML (SAX) [4], that can process large XML documents. Compared to DOM, SAX provides a completely different approach for accessing XML documents. SAX provides a (low-level) token-oriented API to XML documents.

This means that SAX basically works as a lexer/tokenizer, allowing application programmers to implement a series of callback functions that reacts to events fired by a parser. SAX is stream based and does not build any internal representation of the document. This makes SAX better for accessing large documents. SAX does not provide any means of updating XML documents, which leaves apllication programmers very much in their own, when implementing updates.

We also build the dictionary search function with SAX. Implementing callback functions means extending a `DefaultHandler` class and implementing methods for different events. Example 2.3 illustrates this.

The search handler maintains a variable `state` which can be either `KEYWORD` (the element is a keyword element), `FOUND` (the keyword is found) or `FINISHED` (the keyword is found and the result has been processed). Each time a start tag, end tag or character event is fired, the search handler checks the state and takes the appropriate action. Example 2.3 shows callback functions `startElement`, `endElement` and `characters` implemented to search for a keyword in a dictionary. Basically the example maintains a state and correspondingly takes appropriate action. The example searches by comparing a keyword with appropriate (characters that represent keywords in dictionary) characters read (lines 38-40) and when the word is found, a result is buildt using a StringBuffer (lines 13, 25, 42-43, 47).

This method is one often used in SAX applications. As the application logic becomes more complex, the number of possible states rise and the complexity of the SAX application rise.

```
1  private static class SearchHandler extends DefaultHandler{
2    ..
3    private StringBuffer result;
4    private int state;
5    private String keyword;
6
7    ..
8
9    public void startElement(String uri, String localName,
10                            String qName, Attributes attributes){
11     switch( state ){
12       case FINISHED: return;
13       case FOUND: result.append("<" + qName + ">"); return;
14     }
15
16     if( qName.equals("keyword") )
17       state = KEYWORD;
18   }
19
20   public void endElement(String uri, String localName,
21                          String qName){
22     switch( state ){
23     case FINISHED: return;
24     case FOUND:
25       result.append("</" + qName + ">");
26       if(qName.equals("word")){
27         state = FINISHED;
28       }
29       return;
30    }
31    if(qName.equals("keyword")) state = UNKNOWN;
32  }
33
34  public void characters(char[] ch, int start, int length){
35    switch( state ){
36    case FINISHED: return;
37    case KEYWORD:
38      if(keyword.equalsIgnoreCase(String.valueOf(ch,
39                                                 start,
40                                                 length))){
41        state = FOUND;
42        result.append("<word>\n  <keyword>");
43        result.append(ch, start, length);
44      }
45      break;
46      case FOUND:
47        result.append(ch, start, length);
48     }
49   }
50 }
```

Example 2.3: Part of search handler used to search for words in dictionary. The code shows callback functions `startElement`, `endElement` and `characters`. The example is simplified for illustration purposes, hence '..' denotes missing code. Full source code is found in appendix B.

SAX development is more challenging and less intuitive than DOM development, because the API does not provide functionality for accessing the tree structure of XML documents.

The strength of the SAX is its ability to scan and parse gigabytes of XML documents without reaching resource limits, because it does not create a representation in memory of the data being parsed. Due to its design, SAX implementations are generally faster and require fewer resources than DOM.

2.2.3 The Extensible Stylesheet Language

Besides DOM and SAX, another popular approach to programming with XML is using the *Extensible Stylesheet Language*(XSL)[21].

XSL is made up of two parts:

1. *XSL Transformations* (XSLT) [15]
2. *XSL Formatting Object* (XSL-FO)

XSLT is a stylesheet language for defining transformations of XML documents into other XML documents. XSLT-FO is a language for specifying low-level formatting of XML documents [11], and is not covered here.

XSLT is an XML language, i.e. a given XSLT stylesheet (program) is an XML document. XSLT is a declarative language; you state what you want, but not how you want it done. XSLT uses *pattern matching* and *template rules* to perform transformations, as illustrated in example 2.4.

Template rules contain rules to be applied when specified nodes are matched. Template rules identify the nodes to which they apply (they match) by using a pattern, e.g. the template rule `<xsl:template match="/">` will match the root node of any XML document, as the pattern "/" match the root node. Element `<xsl:apply-templates/>` recursively processes children of a matched element. A pattern may be given to specify which children to match, e.g. `<xsl:apply-templates select="keyword"/>` process all children matching the pattern "keyword", i.e. any keyword element.

XSLT is typically used to transform XML documents into XHTML[23] documents which can be rendered by (most) web browsers. The transformation can either be done on clients (by web browsers) or by web servers (using e.g. Apache Xalan [29]).

However XSLT can also be used to extract parts of XML documents and restructured documents. XSLT can extract the needed data and possibly transform it into a new structure.

That XSLT can extract (select) certain parts of an XML document, makes an interesting case. Although it was never designed as a query language, its ability to select and transform pieces from large XML documents makes it usable for querying [30, p.45], though it is not possible to express joins between documents.

The term stylesheet, defined as a document that separates content and logical structure from presentation [11], lacks something to fully define XSLT documents. The term XSLT program is perhaps more suitable, because XSLT can, as mentioned above, express more than just that of a stylesheet.

Because of XSLT's ability to perform queries, searches in the dictionary can easily be performed. Example 2.4 is an XSLT program that selects and returns `word` elements which have a `keyword` with character data content equal to foo.

The example illustrate pattern matching and template rules. Line 3 is a template that match the dictionary element (root element). Line 4 performs a pattern match on templates that match `word` elements with a `keyword` element

```
1  <xsl:stylesheet>
2
3    <xsl:template match="/dictionary">
4      <xsl:apply-templates select="word[./keyword/text()=foo]"/>
5    </xsl:template>
6
7    <xsl:template match="*">
8      <xsl:element name="{name(.)}">
9        <xsl:apply-templates/>
10     </xsl:element>
11   </xsl:template>
12
13 </xsl:stylesheet>
```

Example 2.4: XSLT stylesheet that returns word foo in dictionary. The example is simplified for illustration purposes

with text content foo. Line 7 is a template that match anything and just return the matched element.

The main strength of XSLT is it being designed for performing transformations of XML documents. The declarative style of XSLT allows transformation to be easily expressed.

XSLT can be implemented using different approaches. A straight forward approach would be to implement XSLT on top of DOM[2], i.e. create a DOM tree and traverse this document. This of course implies the problems regarding memory, which was mentioned in 2.2.1.

2.3 Persisting XML

When building applications that use XML as a data source, it becomes necessary to store the data. Real-world applications need data that survives the application's process and data that can be shared between processes. Persisting XML documents means storing documents, in some form, on some external storage device.

A number of approaches are used when persisting XML documents. XML documents can be stored in flat files as described in section 2.3.1. Relational Database Systems may be used to store XML documents, see section 2.3.2. A new type of databases specifically designed to store XML documents is emerging, these are called *Native XML Databases* (NXD) and are introduced in section 2.3.3.

2.3.1 Flat files

Probably the simplest and most commonly used approach to persisting XML documents is to store a serialized version of the XML document in a flat file in the native file system of the operating system. This has the main advantage that it is straightforward and simple. An XML application using flat files to store XML documents will typically follow a certain number of steps:

1. Read the document from the file

[2]Xalan is a XSLT processor for transforming XML documents. It is build using SAX and DOM [29]

2. Parse the serialized representation of the document into some internal (in memory) representation e.g. a DOM tree
3. Manipulate the data by traversing and updating the in-memory representation of the XML document. Updating may build a new in-memory representation
4. Unparse (flatten) the updated XML document and write it to the file system

These steps all have to be implemented by the application programmer and have a number of obvious shortcomings. Parsing the whole document into an in-memory representation naturally limits the size of documents. The approach only supports serial access to the persisted data, i.e. data is read in a stream.

Furthermore, all concurrency issues must be implemented by the application programmer in a multi process environment, making it difficult to implement even simple applications.

2.3.2 Relational Databases

One approach to store XML documents is to store the XML data in relational database systems. Thereby issues such as concurrency control, scalability in multi user environments, data integrity, transaction control and security are maintained by the database.

Using a relational database for storing XML data requires a mapping from the XML data to database relations. Two distinctive mapping techniques are *table based mappings* and *object based mappings*.

Table based mappings maps an XML document to one table or to a set of tables. Consider the XML document and its mapping illustrated in figure 2.3

```
<table>
  <row>
    <col1>info1</col1>
    ..
    <coln>infon</coln>
  </row>
  <row>
    <col1>info1</col1>
    ..
    <coln>infon</coln>
  </row>
</table>
```

```
     table
     -----
col1  ...  coln
----  ---  ----
info1      infon
info1      infon
```

Figure 2.3: Table-based mapping

Table based mappings are simple but only work when the documents are highly structured[31] and therefore only work with a limited subset of XML documents.

The object based mapping is more complex, it models tree objects and then maps these to the database. The objects involved are specific to the XML

document's DTD, i.e. these objects model the data of XML documents and are not to be confused with DOM objects, that model the structure of XML documents[31].

Both mapping types are `bidirectional`, i.e. they can be used to transfer data both from XML documents to the database and from the database to XML documents [31].

XML documents may be divided into *data-centric* and *document-centric* documents. Data-centric documents are well structured and therefore easy to map to relational databases. Data-centric documents often contain repeated structured and are not likely to be targeted for the human reader. An example of data-centric documents are XML RPC implementations such as SOAP [30]. Document-centric documents have an irregular structure and can be considered as real documents, i.e. some text with a logical structure, e.g. the document containing this thesis.

Using relational databases for storing XML data is generally a good choice for data-centric documents and a poor choice for document-centered documents.

In order to initialize the tables in the relation database, the logic structure of the XML document must be known prior to initialization. This require the XML data to be logically described as by DTDs or XML Schemas.

Further, if needs arise for persisting XML data, which do not conform to the existing tables, the database tables need to be expanded and reinitialized (using a new logic description).

When the structure of XML documents is irregular, the result is either a large number of columns filled with null values or a large number of tables [31]. A large number of null values waste space and a large number of tables will lead to slow read and write operations due to the larger number of joins needed for each database query [30, p.31].

XML Enabled Databases

All major database vendors have implemented support for storing XML documents directly in the database. Such support makes a database an XML Enabled Database (*XED*). XML Enabled Databases typically add functionality to a relational database, allowing for retrieval and storage of XML data. This is achieved using the existing functionality of the underlying relational database and the extensional bidirectional functionality for converting XML to relational data and back. As such XEDs are just extensions of relational databases and therefore has the same advantages and shortcomings (see 2.3.2).

2.3.3 Native XML Databases

The most recent advance in database technologies regarding XML is Native XML databases. Native XML databases (NXDs) are designed specifically for storing XML documents. Like other databases, they support features such as concurrency control, scalability in multi-user environments, data integrity, transaction control, security, query languages, etc. The difference is that their internal model is specifically designed for persisting XML.

NXDs have the XML document as their fundamental unit. This means that document order, processing instructions, comments, etc are preserved in opposition to XEDs. For the same reason the existence of DTDs and XML

Schema's are not an issue as well as irregularly structured XML documents are easily stored (again in contrast to XEDs).

That the fundamental unit is the XML document, makes NXDs handle queries for whole documents very fast and therefore useful for persisting document-centric documents.)

Chapter 3

Value-oriented programming

This chapter describes the basic concepts of *value oriented programming* and put forth advantages of this programming model in a distributed environment.

Central to distributed systems are validity of data, replication and atomic updates. A main concern in such systems is to keep data consistent after imperative updates. Upon failures the problem of partly updated data arise, this problem becomes more complex in a distributed setting where data is replicated on different machines. To ensure validity of data, complex transaction mechanisms are necessary as imperative updates are not atomic "by nature". Transaction mechanisms provide atomicity as it ensures an all or nothing update [5, ch. 12-13].

Replication is a key feature in distributed systems as it provides increased performance, increased availability and fault tolerance [5, p.554]. Replication is found throughout distributed systems, e.g. web browsers cache the content of visited web sites, DNS servers replicate domain names to IP mappings to ensure effective and highly available access, etc.

However, replication of updateable data requires coherence protocols to ensure up-to-date data. Such protocols limit the effectiveness of replication [5, p.554]. Replication of immutable data is effective and trivial as no coherence protocol is necessary. With this in mind we introduce a programming model that revolves around immutable data. Such a programming model eases the above mentioned problem of transaction control as data cannot be updated, and offers, where applicable, a possibility to implement a simple light weight transaction control.

The programming model is called value-oriented programming. Value-oriented programming is programming with *values* and *value references*. Values are immutable entities, e.g. the value 5 will always be 5. Value references are references to values and are values themselves. Value-oriented programming adopts a *share-and-create* style (known from functional programming languages such as ML).

First we describe the basic terminology and concepts of value-oriented programming. Then working with trees in a value-oriented fashion is described in section 3.2. We then describe the value-oriented programming model in con-

text of a distributed storage manager. Finally we put forth advantages of this programming model over the traditional imperative model, when building distributed systems.

3.1 Value-oriented concepts

Programming with values may, for the purely imperative programmer, seem alien and not very convenient. As the imperative paradigm revolves around assignment and hence modifying data, not being able to update data may seem like a limitation in the programming model. The imperative programming model has a *copy-update style* of data manipulation. This is illustrated by the following example (using java syntax), creating a stack and adding two entries. The stack is being updated twice, and the values 2 and 1 is copied on each each `push` call.

```
Stack s = new Stack();
s.push(2);
s.push(1);
```

Implementing the same example in a value-oriented context, may seem impossible as values cannot be updated. Another approach must be adopted.

3.1.1 Create

A value oriented version of the example above is done by creating new values, in ML syntax, could look like;

```
val s = [];
val s = [2] :: s;
val s = [1] :: s;
```

In contrast to the imperative version nothing is updated. Each append (`::`) creates a new list, leaving the "original" stack unchanged. A reference to the value is bound to name `s`, three new values are created and their references are bound three times (to the same name, hence the last binding will shadow the first two).

It is not necessary to move to a functional programming language, to find examples of value-oriented concepts. Java's `String` API is value oriented (the String class is final in Java and cannot be inherited). Strings in Java are treated as immutable objects, i.e. all "modifying" methods will create a new String object, leaving the "original" object unchanged. For example the method `concat` which concatenates two `String` objects, by returning a third:

```
String l = "Val";
l = l.concat("ues");
```

This example concatenates two strings by creating a third, leaving both "original" strings unchanged (but left as garbage). All methods in the `String` interface manipulates `String` objects in this fashion.

3.1.2 Share

Previous examples illustrate the create part of the share-create style. The share part is hidden from the programmer. Sharing means keeping only one copy of the same value, and then using references to this value. Sharing values is possible, because values are always *boxed*, i.e. referred to with a reference (box). Figure 3.1 illustrates the difference between boxed and *unboxed* values. In an unboxed representation the value is kept, and in a boxed representation (only) a fixed sized reference (box) to the value is kept. We purposely leave it unspecified where values and references are kept, as this may be in memory, on disk or on a network. This is discussed further in section 3.3.

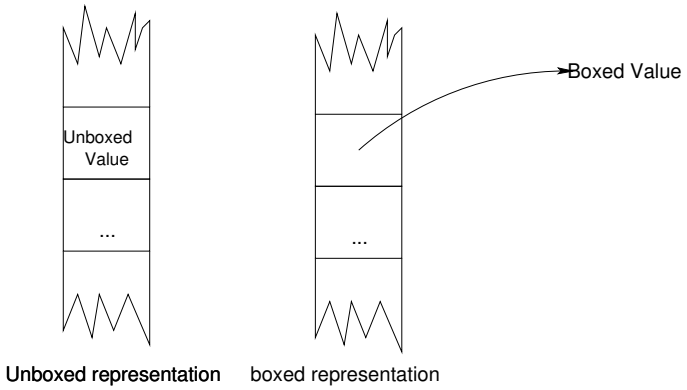


Figure 3.1: Unboxed and boxed representation of a value. In unboxed representation the value is saved directly in the allocated slot. In boxed representation the slot contains a reference to the actual value, which is saved elsewhere.

References and the immutable nature of values enable an efficient sharing scheme. When sharing values, only one stored copy of a given value exists, so when using the value several times, references to the single copy are used. This can be done without any concern to updates of the value (as it is immutable). Further it is possible to copy (cache) a value without any concern for coherence (see section 3.3). This is an important property of why value-oriented programming ought to be feasible in a distributed environment.

The following example creates three lists, where the third list is created by sharing (as opposed to copying) the previous two lists.

```
val l1 = [1, 2, 3];
val l2 = [4, 5, 6];
val l3 = l1 @ l2; (* [1, 2, 3] @ [4, 5, 6] *)
```

Cells

Programming with immutable values will in certain applications and situations result in extensive and complex code. Such situations occur when shared data is updated frequently and updates must be immediately reflected on processes sharing the value.

A programming model that differentiates between values and mutable objects can benefit from this. This can be done by differentiating between references to immutable values and references to mutable data. That way immutable values can still be cached without concerns of validity.

To provide references to frequently updated values the programming model introduces the concept of *cells*. Cells are references to data that may be updated, i.e. unboxed data. Keeping unboxed data directly in a cell has the disadvantage that the size of the data may change, i.e. cells cannot be fixed size. Fixed size cells can hold a reference to data instead.

Cell references differ from box references as they offer the possibility to update the reference to the value they refer to. I.e., they not only support unboxing and boxing operations, they also offer update operation. Figure 3.2 depicts a cell reference.

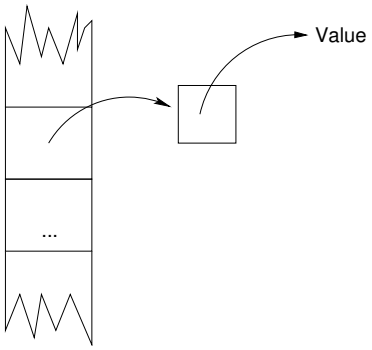


Figure 3.2: A cell representation of a value. The cell holds a reference to a fixed sized slot. This slot holds a reference to the represented value. A cell is updated by writing a new reference (to the updated value) in the fixed sized slot.

3.2 Value-oriented trees

Because XML documents are trees, as described in section 2.1, we describe how to work with trees in a value-oriented context. In value oriented programming all nodes (elements and leaf nodes) are considered values (not considering cells), and child references are value references.

Consider again figure 3.3, as it illustrates a tree representation of some XML data. Subtrees can be shared. When a tree contains values (subtrees), which already reside in another tree, these values are shared.

Figure 3.4 illustrates sharing of subtrees. The substree with root node `type` is shared. Figure 3.4 also shows that when sharing values, XML documents can be thought of as directed acyclic graphs (dags). As subtrees can be shared between an arbitrary number of elements it becomes difficult to keep parents pointers, as each node may have an arbitrary number of parents.

DOM trees are modified imperatively by destructively updating the content of an element, corresponding to only having cells. In value-oriented programming every single element in a tree is an immutable value. This mean that up-

```
<word>
  <keyword>foo</keyword>
  <desc>
    <p>
      <type>jargon</type>
      /foo/ A sample name for
      absolutely anything ...
    </p>
  </desc>
</word>
```

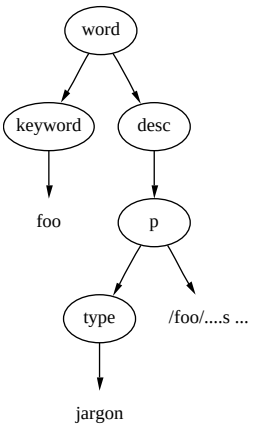


Figure 3.3: The left figure shows an example of XML data. The XML data have no attributes. The right figure shows the corresponding tree representation.

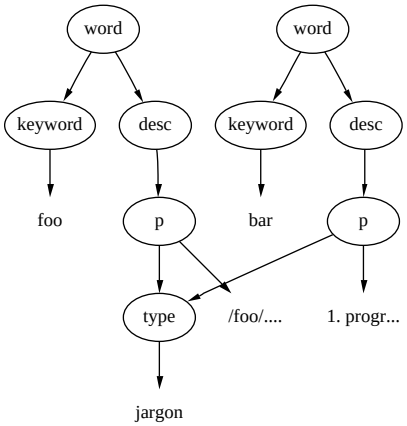


Figure 3.4: Sharing of subtrees, the element type is shared between the two trees

dating a tree will create a new tree, (typically) by sharing parts of the "orginal" tree.

Consider figure 3.5, illustrating the operation of updating the keyword "bar" to "foo". This will create a new `keyword` element (`keyword'`) and a new `word` element (`word'`). The `word'` element shares non updated parts of the old tree (`desc` and `p`). If no other node is referring to the `keyword` element, it becomes garbage. This also illustrates that when modifying elements this way, sharing values can be done without any concern for coherence protocols.

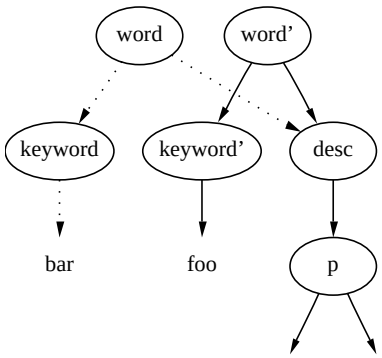


Figure 3.5: "modifying" subtrees means creating new trees. When e.g. the `p` node is changed to `p'` (a new subtree is created), witch propagates all the way to the root node

3.3 Distributed concerns

This section discusses the value-oriented programming model, in a distributed environment. The above section discussed value-oriented programming in a general fashion, no distinction was made between values in memory, on disk and on other computers. The view provided to application programmers should make no distinction of the location of values as well, e.g. the location of values should be transparent to application programmers.

We have discussed boxed and unboxed representation of values in a general fashion. Normally boxing and unboxing happens in memory from stack to heap and from heap to stack, and in a persistent context boxing is equal to saving values and unboxing is equal to loading values[1].

Value references are a necessary part of sharing data and thus a central part of the value-oriented programming model. Value references must be able to be persistent, such that a storage manager can save values and value references.

3.3.1 Value references

References to values may be an address of (or route to) the actual value or it may be some other identifier. The first kind of references are considered *location dependent value references.* and the latter *Location independent value references.*

In a distributed environment this distinction becomes more obvious (and more important). If the reference holds the location of the actual value, we will call such a reference a *locator*. A further distinction are made on locators. A locator residing on the local machine (or process) is called a *local locator*, and must hold information of the value's location on the given machine. A *global*

[1]When saving a value, a byte representation is saved on disk and a reference to the saved bytes is returned. This is similar to making a value boxed. When loading a reference is used to retrieve a specific value. This is the process of unboxing a value.

locator is a locator to a value residing on another machine (or process). Besides holding information of the value location on the remote machine, the locator must hold information of the machine's address. Thus a global locator can be thought of as a global address paired with a machine specific local locator.

However, using locators as references has a number of shortcomings;

1. If values are moved, cached locators become obsolete
2. Problems when buffering values in order to perform a single write on the physical disk and the same problems arise when considering asynchronous write (see section 5.2)

To make the design more general, *location independent value references* are introduced (When nothing else is noted we call location independent value references 'value references'). Location independent value references are:

> *.. universal (in the sense that the same references are used for referring to data in memory, on disk or on the net), location-independent (they identify the data, not the location where the data are stored) and nongenerative (the same data have the same value reference)*[32]

Distinct values are mapped to distinct value references, this makes value references immutable values themselves, they cannot be updated to refer to a different value.

Location independent value references have the following properties:

1. A value reference is a function of the value, $vr = f(v)$
2. f must map to a domain of short strings; e.g. $(0,1)^n$, with $n = 128$
3. f must be effiencently computable
4. f must minimize change of clashes, that is $v \neq v'$, but $f(v) = f(v')$
5. f should be cryptographically strong, i.e. not vulnerable to brute force attacks

Candidates for f is *content hashing* functions. Content hash functions compute a given hash value from the content of a given value. I.e., a given value will always content hash to the same key. An example of a content hash function is e.g. *Message Digest 5* (MD5) [33, p. 272]. Hashing the content of value introduces the problem of *collisions*, that two different value has the same hash value. As noted above in item 4, the chosen function should minimize clashes.

To resolve values from value references, a reference resolver can be used. A reference resolver will, given a location independent value reference, be able to retrieve the actual value. In an in-memory environment an example of a simple reference resolver could be a hash-table, which maps references to values (or memory addresses of actual values).

3.4 Distribued advanteges

Introducing the above mentioned programming model will have the following benefits, compared to an imperative (traditional) model in a distributed environment.

Easy replication : implementing replication and caching becomes easy and effective as no coherence protocol is needed. Replication involves caching, memorization and actual copying of values to ensure availability and performance.

Sharing of values : the value oriented programming model provides sharing of values. In contrast to replication, sharing means only holding a single copy of a given value. Values can be effectively shared as they are immutable and never become invalid.

Atomic updates : The value oriented programming model provides atomic updates, as all values are referred to by a reference and values themselves are never updated. Cell updates means updating a cell with a value reference for the "new" value. Updating a value reference can be done atomically. It is possible to implement a simple light weight transaction control. As the "original" value is never changed, a rollback can be implemented simply by changing the reference back to the "original" value.

Chapter 4

The Document Value Model

The DVM is a high level tree-oriented application programming interface (API) for well-formed XML documents. It provides an abstract model for representing XML document structures and specifies methods for creating, accessing, modifying and persisting documents. The interface is value-oriented, i.e. entities in the Document Value Model are considered values.

As described in section 3.1, documents are modeled as directed acyclic graphs (dags) in a value-oriented programing model. We will refer to the document structure as being tree-oriented (trees are also dags).

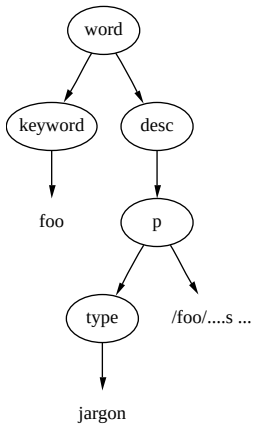


Figure 4.1: Graphical representation of the XML document from figure 3.3 in the DVM. Ellipses denote element nodes and simple text denote character data nodes. None of the element nodes contain attributes.

The API consists of several interfaces, representing entities of documents.

The central interface is `Node`. This interface represents the nodes in a document. Access to XML documents is acquired through `Node`. The interface `MutNode` is a subtype of `Node` and represents mutable nodes. These are introduced to model the cell references introduced in chapter 3. Interface `XMLStoreFactory` is used to create nodes (XML documents). Interface

`XMLStore` is used to load and saved persisting XML Documents.

The API contains basic functionality for working with XML documents. Additional and more convenient functionality can be buildt using the basic functionality.

Section 4.1 describes how `Node` and interfaces related to this interface are used to access documents. The following sections 4.2 and 4.3 describe mutable nodes and a how the visitor pattern [10, p.331] can be used to extend `Node` functionality.

Section 4.4 describes the `XMLStoreFactory` interface for creating XML documents. The `XMLStore` interface is described in section 4.6. Naming documents is described in section 4.7.1. A utility library is presented in section 4.8.

4.1 Nodes - access to documents

The `Node` interface is the primary data type of the Document Value Model. This and interface `ChildNodes` are the basic interfaces for representing XML Documents.

XML elements and XML character data are both represented by the `Node` interface. The interface contains functionality for accessing data of the given element / character data.

The interface `ChildNodes` represents sub elements of an XML element. Attributes and their values are represented as character data using Java's `String` class.

The `Node` interface represents both XML elements and XML character data a node *type* is used to determine if an element or character data is represented. The different values of type are respectively `ELEMENT` and `CHARDATA`.

Nodes have a *node value*. Node value of `ELEMENT` nodes is tag name of the represented element. Node value of `CHARDATA` nodes is that of character data.

Most methods in the interface have different meanings according to the node type. The Node interface is defined as:

`byte ELEMENT`
: Byte value used when `Node` represents an XML element.

`byte CHARDATA`
: Byte value used when `Node` represents XML character data.

`bool isMutable()`
: Returns true when node is mutable. (Section 4.2 describes mutable nodes).

`byte getType()`
: Returns the type of `Node`, either `ELEMENT` or `CHARDATA`.

`String getNodeValue()`
: When nodes have type `ELEMENT`, the name of the element is returned. When nodes have type `CHARDATA`, the character data is returned.

`String getAttribute(String name)`
: Returns the attribute with the given name. Nodes of type `CHARDATA` returns null.

`String[] getAttributeNames()`
: Returns an array of attribute names. Node of type `CHARDATA` return null.

`ChildNodes getChildNodes()`
: Returns children of Node. Nodes of type `CHARDATA` returns an empty `ChildNodes` instance.

The interface `ChildNodes` provides an abstraction of an ordered collection of nodes. Nodes in `ChildNodes` are accessible via an integral index, starting from 0.

Methods in `ChildNodes`:

`int getLength()`
: Returns the number of nodes in the list.

`Node getNode(int index)`
: Returns node with index `index`.

Example 4.1 illustrates how the interfaces can be used to access information in an XML document. The method `void toHtml(Node node, Writer out)` writes a XML document representing a `"word"` node to a character stream as a *Hyper Text Markup Language* (HTML) representation of the word. (HTML is a standard format for representing hypertext on the World Wide Web [34]) The method takes a `"word"` node and recursively calls itself with the children within the `"word"` node.

The example illustrates how data in XML documents is accessed in the Document Value Model.

In the value-oriented model described in chapter 3 child nodes are obtained by using value references for the child nodes. In DVM this is hidden from application programmers to provide a more convenient and usable interface. Child nodes are obtained through the method `getChildNodes()` and the interface `ChildNodes`.

The interface `Node` is used for representing both XML elements and character data. The method `getType()` allows programmers to differentiate between node types.

Another approach could be to introduce sub interfaces representing elements and character data. Such a solution may lead to excessive downcasts, which are unsafe and slow.

4.2 Mutable nodes

The interface `MutNode` represents cells. This interface represents *mutable nodes*, i.e. nodes, which *state* can be updated.

The state is the node value, the attributes and the children of the node.

Method definitions:

`void setNodeState(Node node)`
: The method takes a node as parameter representing the new state of the mutable node.

`Node getNodeState()`
: The method returns an immutable node representing the nodes state.

```
public void toHtml( Node node, Writer out ){
  switch (node.getType()){
  case Node.ELEMENT:
    if(node.getNodeValue().equals("word")){
      out.write("<html><body>");
      toHtml(node.getChildNodes().getNode(0));
      toHtml(node.getChildNodes().getNode(1));
      out.write("</html></body>");
    }else if(node.getNodeValue().equals("keyword")){
      out.write("<b>");
      toHtml(node.getChildNodes().getNode(0));
      out.write("</b>");
    }
    else if(node.getNodeValue().equals("p")){
      out.write("<p>");
      toHtml(node.getChildNodes().getNode(0));
      out.write("</p>");
    }
    else {
      ChildNodes children = node.getChildNodes();
      for(int i = 0; i < children.getLength(); i++){
        toHtml(children.getNode(i));
      }
    }
    break;
  case Node.CHARDATA:
    out.write(node.getNodeValue());
  }
}
```

Example 4.1: The method converts a fragment of an XML document, which represents a ‘word” element, to HTML.

Mutable nodes are usable in situations where documents contain mainly static data, but a small part of the document is frequently updated. When data does not change frequently, mutable nodes should not even be considered, as all advantages of value-oriented programming are lost (see section 3.1). Different examples illustrate usage patterns of immutable nodes.

The dictionary example used throughout the report can be extended to contain information of most popular words. In order to provide this information, a counter can be attached to each word to keep track of how many times the particular word has been requested. The structure of a word subtree is then modified as shown in figure 4.2 to contain a 'count' element node, which contains the 'request count' in a character data node.

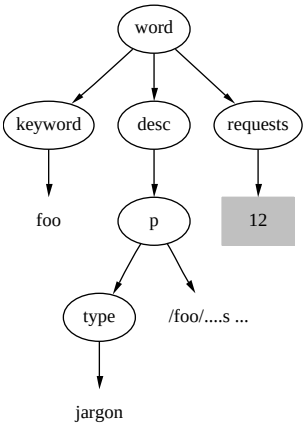


Figure 4.2: The 'word' nodes are modified to contain a search counter. The character data node which contains the count is mutable. This example shows a subtree representing the word 'foo' which have been found 12 times.

In this case it would be cumbersome to update the whole 'dictionary' tree for each successful search. Instead the character data node representing the search count can be made mutable. Example 4.2 illustrates how this may be implemented. (The example is a modification of `XMLStoreDictionary`'s search functionality given in appendix B).

Invoking the methods `getX()` from the `Node` interface on mutable nodes should be done with care. Consider a case where the methods `getAttributeNames()` and `getAttribute(String name)` are invoked on a mutable node. In case another process invokes `setNodeState(Node state)` and change the attributes in between these two calls, problems may occur. Access to the mutable nodes is thus safest through use of the method `getNodeState()`.

Compared to the value-oriented programming model introduced in chapter 3, the `setNodeState` method corresponds to updating a cell reference. The `MutNode` interface thus preserves the property that updates are done atomically.

```
public Node keywordSearch( String keyword ){
  loadDict();
  Document doc = builder.newDocument();
  if( binarySearch(keyword) ){
    NodeList nlst = ((Element)match).getElementsByTagName("count");
    Element count = (Element)nlst.item(0);
    int c = Integer.parseInt(count.getAttribute("value"));
    count.setAttribute("value", c+"");
    saveDict();

    Node n = doc.importNode( match, true );
    doc.appendChild(n);
    dict = null;
    return doc;
  }
  Node res = doc.createElement("word");
  res.appendChild(doc.createTextNode("No match on keyword "
                                     + keyword));
  doc.appendChild(res);
  dict = null;
  return doc;
}
```

Example 4.2: A request for a word is performed. The search is done by the method `binaryLookup`, which is not shown. If the search is successful the search counter of the found word is incremented, and the subtree representing the word is returned. Exception handling is purposely left out to keep code clean and readable.

4.3 Adding functionality

Application programmers may need additional functionality when working with DVM.

In object oriented languages functionality could be added by extending the `Node` interface. This approach have two shortcomings:

- Expensive downcasts required to gain access to extended functionality.
- Many different interfaces, all with different added functionality may be created, or the one additional interface might be populated with many unrelated functions. In any case this makes code complex and inflexible

Applying the Visitor pattern ([10, p. 331]) allows functionality to be added to nodes, without modifying the node interfaces[1].

The Visitor pattern is introduced by the interface `Visitor` and the `accept` method in the `Node` interface. The `accept` method is already described in section 4.1. The vistor interface introduces visit methods take either an immutable node or a mutable node as parameter. The code for visiting a node is written within this method.

When working with the visitor pattern, two approaches for traversing the object structures exist:

1. The traversal code is written in the `accept` method of the object type to be traversed. This has the advantage that traversal code is only written once.
2. The traversal code is written by the application programmer in the `visit` method. The advantage of this approach is greater flexibility, since the traversal can be stopped within the `visit` method.

The Document Value Model adopts the latter approach.

Example 4.3 illustrates how to write a visitor for a `Node`. The example illustrates the same as example 4.1. Code for the `visit(DVMMutNode)` method is not illustrated as the `visit` method in this case contains similar functionality.

The visitor pattern allows flexibility when implementing additional functionality.

4.4 Creation of documents

Functionality for creating new XML documents is specified. This functionality should not reveal the specific implementation of the Document Value Model (DVM).

The Abstract Factory pattern [10, p.87] is used to specify methods for creating nodes, child nodes and instances of the `XMLStore` interface described in section 4.6.

The Abstract Factory pattern hides concrete instantiation (constructor calls), and enables flexiblity in the implementation of the DVM interface, as it can be exchanged without affecting the existing DVM applications.

The create methods are described as:

[1] The visitor pattern provides many of the advantages of high order functions, know from functional programming langauges

```
public class HTMLVisitor extends Visitor{
  private Writer out;
  public HTMLVisitor(Writer out){this.out = out;}

  public void visit(DVMImmNode node){
    switch(node.getType()){
    case Node.ELEMENT:
      if(node.getNodeValue().equals("keyword")){
        out.write("<b>");
        node.getChildNodes().getNode(0).accept(this);
        out.write("</b>");
      }
      else if(node.getNodeValue().equals("p")){
        out.write("<p>");
        node.getChildNodes().getNode(0).accept(this);
        out.write("<p>");
      }
      else {
        ChildNodes children = node.getChildNodes();
        for(int i = 0; i < children.getLength(); i++){
          children.getNode(i).accept(this);
        }
      }
      break;

    case Node.CHARDATA:
      out.write(node.getNodeValue());
    }
  }
}
```

Example 4.3: A Visitor for printing 'word' nodes into HTML factions. The words (keyword) are written in bold and p elements are written as HTML paragraphs. All other element are not converted to HTML but simply traversed to reach the character data nodes.

`Node createCharDataNode(String value)`
The method creates an immutable node representing the character data provided in the argument.

`Node createElementNode(String value, Attribute[] attrs, Node[] children)`
Creates immutable element nodes, given tag name, attributes and children.

`Node createMutCharDataNode(String value)`
Create and return a mutable node representing character data.

`Node createMutElementNode(String value, Attribute[] attrs, Node[] children)`
Create and return a mutable node representing an XML element.

`Attribute createAttribute(String name, String value)`
Creates an attribute, given name and value.

`XMLStore createXMLStore(String name)`
Creates an `XMLStore` instance, given the name for the XML Store.

The `Attribute` interface is only used when initializing nodes. It represents the name and value of attributes.

`String getValue()`
The method returns the value of the attribute.

`String getName()`
The method returns the name of the attribute.

The following example illustrates creation of nodes. An XML element with the tag name "keyword" is created. The content of the element is character data "bar".

```
Node barValue = factory.createCharDataNode("bar");
Node barKeyword = factory.createElementNode("keyword", barValue);
... // create rest of tree
Node word = factory.createElementNode("word",
              new Node[]{ barKeyword, desc })
```

Example 4.4: `createX()` methods are used to create new nodes.

When creating new nodes, application programmers may use existing nodes, and thereby share nodes. Nodes can be shared within the same document or between documents.

Example 4.5 illustrates sharing. The example continues example 4.4. A node with children `fooValue` and `barValue` is created. The node `barValue` is shared.

Shared nodes may be located on disk or on another machine in the network.

```
Node fooValue = factory.createCharDataNode("foo");
Node keywords = factory.createElementNode(
                  factory.createChildNodes(
                    new Node[]{fooValue, barValue}
                  )
                );
```

Example 4.5: Creating Nodes using existing node, from example 4.4

4.5 Modification of documents

Modification of nodes are made by creating new nodes.

New nodes are created by applying the `createX` methods from the `XMLStore-Factory` interface. Node values, attributes and/or child nodes can be reused when creating new nodes.

Example 4.6 illustrates modification. The content of a "keyword" node is changed from 'bar' to 'foo'.

```
Node fooKeyword = factory.createCharDataNode("foo");
Node desc = word.getChildNode().getNode(1);
word = factory.createElementNode("word",
         new Node[]{fooKeyword, desc});
```

Example 4.6: Modifications are done by building new documents. Example change keyword "bar" to "foo", in node "word". The example is a continuation of example 4.4 and 4.5

This style of modifying nodes (values) corresponds to the style described in section 3.2 for modifying value-oriented trees (directed acyclic graphs).

4.6 Persistence of documents

When working with XML documents it is nessasary to load and save documents from and to disk.

Persistence of XML documents is location and distribution transparent. The persistence functionality is provided by the `save` and `load` methods of the `XMLStore` interface.

`ValueReference` is an interface representing location independent value references (as defined in chapter 3).

Methods `save` and `load` are defined as:

`ValueReference save(Node node)`
: Saves documents. `node` is the document root node. A value reference for the persisted document is returned.

`Node load(ValueReference ref)`
: Loads a document using the value reference `ref`.

Value reference are used to retrieve documents and returned when documents are stored. They are unique to saved document as described in section 3.1. The content of the value references is not accessible, not readable and not (re)producible. Application programmers can therefore not produce value references in order to retrieve documents from `XMLStore`.

Example 4.7 demonstrates loading a document and saving a document. How the value reference for the document is obtained is not shown (line 1). When saving the modified document, a new value reference is returned. For the simplicity of the example, code modifying the dictionary is not shown, but is represented by "`...`".

```
1     ValueReference ref = ...
2     Node dictionary = xmlstore.load(ref);
3     ...
4     Node newDictionary = ... //create or modify document
5     ValueReference newRef = xmlstore.save(newDictionary);
6
```

Example 4.7: loading, modifying and saving a XML document.

Using value references when saving and loading XML documents, the `XMLStore` interface provides a location and distribution transparent interface for loading and saving documents.

Mobility of documents can be implemented in DVM without any side effects on the application code.

Besides being location independent, the interface does not reveal how documents are persisted. It provides the application programmer with a tree view of the XML documents.

Another advantage of the `XMLStore` interface is that documents do not have to be loaded fully into memory, i.e. the interface does not specify methods for parsing a whole document into memory, before access and manipulation of the document can be made.

4.7 Symbolic names for documents

For convenience, application programmers should be able to associate documents with human readable names rather than value references. Clients cannot share particular resources managed by a computer systems unless they name them consistently. Names facilitate communication and sharing [5].

The entire world is not value-oriented, e.g. new articles are published by (online) newspapers and the weather forecast change. Consider newspapers published on the Internet. The most recent version of the newspapers is normally retrieved by using a shared updateable name, such as `http://www.politiken.dk`. Sharing documents using DVM, we need to provide a `name service`, allowing retrieval of documents using human readable names.

Before describing a name service in DVM the concepts of `names` and a `name service` are defined. The name service functionality of Document Value Model is defined in section 4.7.2.

4.7.1 Names and name service

A name is a (preferably human readable) sequence of characters, which belongs to a *name space*. A name space is a collection of valid names. Associations between names and resources are called *bindings*. A name service contains zero or more bindings. A name service must be able to create new bindings and resolve names (i.e. look up resources given a name).

Names are said to be *pure* if they contain no location information. *Non-pure* names contain some degree of location information of the resource, which they name. *Addresses* are names consisting entirely of location information.

Value references can be regarded as pure names, who are not human readable. Locator are regarded addresses. Value references are, as described not adequate for retrieving and storing XML documents in the Document Value Model as they are not human readable.

4.7.2 The DVM name service functionality

In the Document Value Model (DVM) XML Documents may be associated with human readable names by creating name to value reference bindings.

The name service functionality of the Document Value Model is contained in the `XMLStore` interface. Using this functionality names are resolved (their associated value reference looked up), new name to value reference bindings are made and existing name to value reference bindings are rebound.

The name service functionality is provided by methods `lookup`, `bind` and `rebind`. They are defined as:

`void bind(String name, ValueReference ref)`
: Creates a binding between `name` and `ref`. The binding is shared with all other peers within the XML Store.

`ValueReference lookup(String name)`
: Resolves `name`, i.e. looks up a value reference. If `name` does not exist, `null` is returned.

`void rebind(String name, ValueReference ref)`
: Updates a name-value reference binding, i.e. after having invoked `rebind(name, valref)`, `name` is mapped to `valref`.

Example 4.8 illustrates how an XML document is retrieved using a symbolic human readable name. The example is an extension of example 4.7. The XML document being loaded and saved is the FOLDOC dictionary (see appendix B). The document name is `"FOLDOC"`.

The name service makes it possible to retrieve XML documents using human readable names in contrast to only using value references. Since value references are location independent but uniquely identifies documents, a pure name space is obtained, in which document names are independent from document locations.

The name service functionality introduces an imperative aspect in DVM due to the destructive update functionality of the `rebind` method. Each time `rebind` is used a value reference is removed from the name service. Documents obtained through the removed value references are thus not retrievable using symbolic names.

```
ValueReference ref = xmlstore.lookup("FOLDOC");
Node dictionary = xmlstore.load(ref);
...
Node newDictionary = ... //create or modify document
ValueReference newRef = xmlstore.save(newDictionary);
xmlstore.rebind("FOLDOC", newRef);
```

Example 4.8: The XML document with the symbolic name `"FOLDOC"` is loaded, modified and saved. First the symbolic name is resolved and next the obtained value reference is used for loading the document. After saving the modified document the symbolic name is updated with the new value reference.

As the number of symbolic names in a name service increases it becomes more difficult to come up with new document names. Application programmers wanting to name their documents might thus choose names already used. This makes the name service insufficient for storing huge amounts of bindings. A more complex solution providing a more suitable name service functionality is required.

4.8 Utility library

The Document Value Model provides basic functionality for accessing and manipulating XML documents. Writing applications using this Application Programming Interface (API) may become tedious and inconvenient.

A *utility library* containing extra functionality has been developed to make the API more usable.

The provided utility library has been implemented during development and testing of a DVM prototype implementation. Only a subset of utilities are described. The utility library can be separated in two groups: methods for convenient access to and modification of nodes and methods for building DVM representations from different XML representations, e.g. from serialized XML documents.

The first group of methods (access and modification) are described in section 4.8.1 and the last group in section 4.8.2.

4.8.1 Access and modification

The utility library methods for access and modification is specified in class `DVMUtil`. A subset of these is presented below:

`Node replaceChild(Node n, int index, Node child)`
: Returns a new node, created by replacing `child` with `n`'s child node at index `index`.

`ChildNodes replaceChild(ChildNodes children, int index, Node n)`
: Returns new ChildNodes, created by replacing node at `index` in `children` with node `n`.

`Node insertChild(Node n, int index, Node child)`
Returns a new node, created by adding node `child` to index `index` in `n`'s child nodes.

`Node removeChild(Node n, int index)`
Returns a new node, created by removing child node at index `index` in node `n`'s child nodes.

`ChildNodes subChildNodes(ChildNodes childNodes, in start, int length)`
Returns a ChildNodes, which is a sub list of `childNodes`. The sub list begins at index *start* and ends at index *start + length*.

`Attribute[] getAttributes(Node n)`
Returns an array of the attributes, associated to the node `n`.

`boolean equals(Node node1, Node node2)`
Checks equality of two nodes. In order to be equal two nodes must have the same type. For character data nodes, the character data (i.e. node values) must also be equal. For element nodes equality is checked by

1. equality of the tag names (i.e. node values).
2. equality of the attributes. The same attributes must occur within the elements, and each attribute must have the same value within the elements.
3. equality of child nodes. Child nodes must occur with the same order in the elements. Their equality is checked recursively.

(Another approach would be to test equality by comparing value references for the nodes. This approach is however not applied.)

Example 4.9 shows removal and insertion using the methods described above. The example is a modification of example 4.5.

```
word = DVMUtil.remove(word, 0);
word = DVMUtil.insert(word, 0, fooValue);
```

Example 4.9: Utility methods `removeChild` and `insertChild` makes DVM more convenient to use.

Example 4.9 purposely illustrates removal and insertion using utility methods `removeChild` and `insertChild`. This is illustrated in example 4.10.

```
word = DVMUtil.replaceChild(word, 0, foo);
```

Example 4.10: The method `replaceChild` provides functionality for replacing child nodes.

The introduced utility methods improve the programming using DVM. Similar methods can be found in the Document Object Model. The semantics of the interfaces however differ significantly, as destructive updates are not performed in DVM. Instead the utility methods perform updates by creating and returning new nodes.

4.8.2 Building DVM representations

While developing and testing the prototype implementation, functionality for building XML documents persisted as flat files into a DVM representation was necessary. Such functionality would be of general use, especially when users of applications must write small XML documents.

This functionality is part of the utility library and implemented in the class `DVMBuilder`. The methods of `DVMBuilder` are described as:

`Node parse(InputSource in)`
: Reads an XML document in serialized form, from an `InputSource` and builds a DVM representation of the document. The return value is the root node of the DVM representation. The DVM document consists of immutable nodes and is not saved.

`ValueReference unparse(InputSource in)`
: Unparses an XML document in serialized format to a persistent representation in the Document Value Model (DVM). The DVM document consisting of immutable nodes is saved during unparsing. A reference to the document's root node is returned.

Example 4.11 illustrates how the `unparse` method is used to unparse an XML document in serialized representation into a persistent DVM representation.

```
InputSource in = new InputSource( new FileReader( xmlFile ) );
XMLStore xmlstore = ... // initialize xmlstore if not already done
DVMBuilder builder = new DVMBuilder( xmlstore );
ValueReference ref = builder.unparse( in );
```

Example 4.11: DVMBuilder is used to unparse XML persisted in a flat file into a persistent DVM representation. The method returns a reference to the root node of the XML document.

The `unparse` method is useful in situations, where XML documents persisted in flat files must be persisted in a DVM representation.

4.8.3 Summary

The introduced Document Value Model (DVM) has a value-oriented interface for creating, accessing, modifying and persisting XML documents. DVM has the following properties:

- DVM does not offer a vast and complex interface for working with XML documents, only basic functionality is specified.
- The DVM functionality is extensible, either by inheritance, development of utility libraries or by using the provided visitor pattern.
- DVM offers no destructive update operations on the `Node` interface. Modifications are either performed by creating new `Node`'s or by using the `MutNode` interface, which corresponds to the use of cells as described in chapter 3.

- The interface allows convenient traversal of document trees, as references are not needed to retrieve child nodes.
- Nodes may be shared.
- Location and distribution of documents are transparent to the application programmer.
- DVM allows lazy loading of documents (on request). This prevents loading whole documents into memory, when only a small part is needed.
- In-memory nodes and on-disk nodes are treated equally by application programmers.

Chapter 5

XML Store architecture

An implementation of the Document Value Model (DVM) described in chapter 4 is a storage manager that handles persistence and distribution. This chapter describes the design of such a storage manager, called *XML Store*.

XML Store is a value-oriented storage facility that transparently persists and distributes XML documents. XML Store supplies the Document Value Model, allowing application programmers to persist, access and manipulate XML documents stored in an XML Store (network).

XML Store is a peer-2-peer storage facility (as defined in section 1.5), the term XML Store refers to a peer-2-peer network. The term *XML Store peer* refers a single peer (computer) in the network. Each XML Store peer has functionality to persist XML documents and to retrieve XML documents stored in XML Store. Figure 5.1 illustrates an XML Store. No central servers exist and each peer therefore acts both as a server and a client.

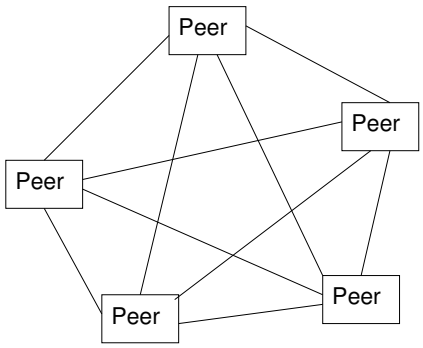


Figure 5.1: XML Store consists of XML Store peers. Each peer may communicate with all other peers in the network.

To load documents stored in XML Store, peers use *value references* as described in chapter 3. Value references to all nodes in any XML document exists, as described in section 3.2. To load a document from another peer, its location in the network (ip and port) must be known. As described in section 3.3.1, value references are location independent. To resolve a value (node) from a value reference, a locator is needed. Locators are, as described in section 3.3.1,

an address of the actual value. The process of loading a document given a value reference therefore includes retrieving a locator for the document (node). Retrieving a locator is done using a *reference server*. The reference server can conceptually be thought of as a global hash table that contains value reference to locator mappings. The reference server is part of the peer-2-peer system, i.e. distributed among XML Store peers. Communication with other reference server peers is done using *IP Multicast*[5, p. 154] as described in section 5.1.

With a locator a connection to the peer holding the document can be made. This connection is made using sockets.

XML Store peers have a layered architecture. They consist of a DVM layer and a disk layer. Figure 5.2 depicts the layered structure of a peer. The disk layer provides functionality for saving and retrieving data in the form of bytes and is described in section 5.2.

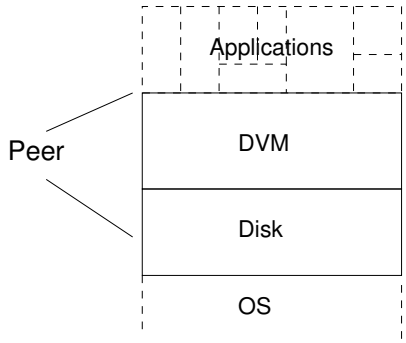


Figure 5.2: Single peer layered architecture consisting of a DVM layer and a Disk layer. Applications are build on top of the DVM layer.

To allow application programmers to give XML documents stored in XML Store symbolic names, a name server functionality is necessary. The name server maps symbolic names to value references, e.g. "dictionary" → "some 128 bit location independent value reference". The name server should also be part of the peer-2-peer network, but is in the prototype implementation given with this thesis, a simple insufficient client-server solution.

These concepts *disk*, *DVM*, *reference server* and *name server*, are the basic concepts of XML Store. The disk layer consists of two parts, the actual disk part (handles loading and saving from physical disk) and a reference server part. The reference server is integrated with the disk, the disk uses a reference server when loading and saving. The name server is in the same way part of the DVM layer. Application programmers use the DVM interface, parts of this include binding, unbinding and looking up symbolic names, this is handled by the name server. This is illustrated in figure 5.3 that illuterates steps involved in retrieving a document from an XML Store.

The process of retrieving documents in XML Store involves several steps:

1. given a document name a value reference to the document is looked up using a name server.

2. loading the document using this reference involves:

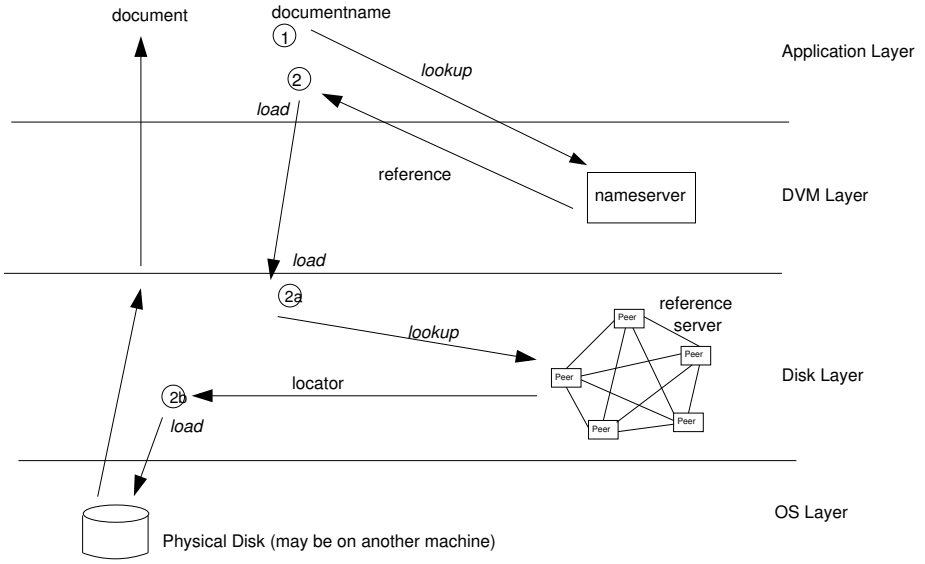


Figure 5.3: Retrieving a document from XML Store. The value reference associated with a document name must be looked up using a name server. To retrieve the document from the value, a locator must be looked up using a reference server. Given a locator the actual document can be loaded. (*Italic* text denotes actions.)

(a) using the value reference server to lookup a locator.

(b) using this locator loading the actual document.

(c) returning the loaded document.

Documents are loaded lazily (to prevent whole documents from being loaded into main memory), thus loading a document means loading the root node. When child nodes are needed, these are loaded using the same process but starting at point 2a.

The reference server is described in section 5.1. The disk functionality is described in section 5.2. The name server is described in section 5.3. The DVM interface is described fully in chapter 4.

5.1 Reference server

The reference server provides a distributed service for mapping value references to locators, needed to locate a value (node), in XML Store. The service is basically a distributed hash table allowing key (value reference) to data (locator) mappings.

The distributed hash table should conform to these requirements:

Decentralized Scalability is a requirement (see section 1.1.1) and since centralized solutions introduces possible bottlenecks in distributed networks

(and therefore is not scalable), a decentralized peer-2-peer solution with a network of reference server peers is preferable.

Flat network hierarchy All peers in the distributed reference server network contribute on equal terms. They all have responsibility to persist mappings, and none of them are assigned any key role with a greater responsibility than others.

Stateless network peers A peer in the reference service network keeps no explicit knowledge of other peers. This in order to conform with the requirement that no XML Store peer keeps knowledge of any other peers in an XML Store network. This makes it possible for peers to join a network without the updating existing peers with its presence.

Persistent mappings Two properties of the XML Store require persistent mappings.

1. Mappings should survive the termination of processes. Mappings residing in dynamic memory are not reproduceable after process termination. Persistent mappings are reproduceable.

The reference service network is constructed, such that a part of each XML Store peer is a reference service as illustrated in figure 5.4.

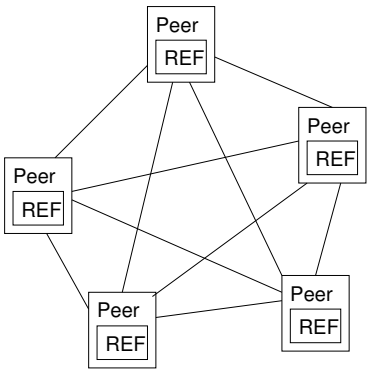


Figure 5.4: The XML Store peer-2-peer network. Each XML Store peer contains reference service functionality (denoted by REF). Communication between the peers takes place to provide a distributed reference service.

A reference service peer is constructed as an API which is usable without knowledge of XML Store peers. This way the functionality can be used by other applications (although developed specifically to be used in the XML Store network) and the implementation of XML Store peers and reference server peer can change independently.

The reference server provides basic functionality of a hash table (lookup, bind).

`Locator lookup( ValueReference vr )`
A locator is looked up, using value reference *vr*.

`void bind( ValueReference vr, Locator loc )`
The given locator is bound to the given value reference.

5.1.1 Protocol

Building a decentralized distributed hash table functionality requires interaction between participants in the network.

The *reference service protocol* is the protocol for performing a lookup and bind in the distributed decentralized reference service network.

As described in section 5.1 reference service peers do not contain information of other peers. The reference service protocol must thus be performed without any knowledge of participating peers.

IP Multicast allows communication without knowledge of participants in the lookup service network. IP Multicast allows a sender to send IP packages to a group of receivers, without knowing the identity of those receivers. Such a group is called a *multicast group* and is specified by a *Class D Internet address*, i.e. an Internet address used specifically for multicast groups [5, p.93]. Receivers has to join the group using the same Class D Internet address. IP Multicast is buildt on top of the Internet Protocol (IP).

Using IP Multicast the following protocol is used to perform distributed lookup:

1. The requested key (value reference) is looked up locally. If the key resides locally, its associated data (locator) is returned.

2. If the key does not reside locally, a lookup request is sent to the multicast group. The lookup request contains a *lookup request id* and the key. A request id uniquely identifies the lookup request.

3. When a peer from the multicast group receives a lookup request, it checks if it holds the key-data mapping. If it does, it replies directly to the lookup peer having sent the request. The reply is a *lookup answer* which contains the lookup request id and the requested data.

   If the peer does not hold the key-data mapping no further action is taken. This will eventually lead to at timeout, if no peer holds the key.

4. The requesting peer caches the mapping when it receives a reply.

As described above key-data mappings are always persisted locally when being bound. The bind operation does not check if the key-data pair resides on other reference service peers, before binding the value. It thus consists of one operations:

1. The key-data mapping is persisted locally.

The bind functionality introduces a possibility that keys are bound to different data, e.g. a key may be bound different data on different reference service peers. However, the reference server binds value reference to locator mappings, which are used to resolve values. As there always exists only one value to a given value reference (see section 3.3.1), this poses no problem. If a value reference is bound to two different locators, any one of these can resolve the value.

IP Multicast is not a reliable service, as there is no guarantee, that a message reach all members of the multicast group. This may introduce faults in the lookup service functionality. However, a reliable multicast can be implemented as described in Coulouris et al. [5, p. 439] and as done in javagroups [**?**].

5.1.2 Persistent hash table

The reference service must be fault tolerant. As described in section 5.1 this is achieved by persisting the key-data mappings.

A persistent *hash table* is used to map keys to data entries.

The persistent hash table keeps all mappings both memory and persist them on disk. New key-data mappings are written to disk, when performing bind operations.

Keeping mappings in memory improves the lookup performance as disk is not accessed. The disadvantage is that large quantities of mappings may be kept in memory. This can be solved using a sophisticated persistent datastructure.

5.2 Disk

The lowest layer in the XML Store architecture is the disk layer. The disk layer handles actual persistence of values.

An XML Store disk manages low-level loading and saving of bytes. An XML Store disk is a software representation of a physical disk. Thus creation and deletion of disks can be done dynamically. (Since disks are actual areas of static memory, storage area might be a more correct name for disks). Unlike conventional disk (in e.g. an operating system) the disk layer has the following properties.

No addressing When saving a value on disk, the client (application programmer) does not specify where the value should be located. I.e., application programmers cannot save values at a particular address. Instead the disk will decide where values are saved and return a value reference. This allows a simple storage strategy in which values are saved log-structured or sequentially. Saving values log-structured does not require random write access, as values always are saved at the end of the log [35]. Loading values requires random read access, as loading is not necessarily done sequentially but from random locations.

No deletion Deleting stored values is not possible. Disk offer no support for erasing values, because data may be (transparently) replicated, "deleting a value" has no meaning. Furthermore, deleting values poses a danger of dangling references when multiple references exists.

Thus disks can only be filled with values and consequently their space limits will be reached (otherwise disks would represent infinite storage units). Therefore garbage-collection is needed to prevent disks from being filled with values no longer reachable (live). Garbage-collection is beyond the scope of this thesis.

Sharing values As disks are used to store immutable values, values can be easily shared between disks. A given value must only be saved on disk once. The second (and the following) times a client saves a given value nothing is written on disk, but the reference to the existing value is returned.

Sharing is only interesting on a single peer. When multiple disks exists on multiple computers, repeated distributed loadings of the same value is not feasible, as distributed loading degrades performance. Applying a proper strategy for caching distributed loaded values improves performance.

Configurable and extensible Disks can be configured arbitrarily and new features can be added without modifying existing code. Disks may support different features, such as buffering, caching, asynchronous read/write, global accessibility (such that other disks can load values saved here) etc. Each disk may have its own unique set of features, configured by the application programmer. In Java this is implemented using the Decorator pattern [10, p.175], such that disks can be created by combining different disk and thereby add different features. Examples are

```
Disk d = new BufferedDisk(new CachedDisk(new LocalDisk());
```

or

```
Disk d = new CachedDisk(new GlobalDisk(new LocalDisk(),port));
```

This simple design also makes disks extensible as new features may be added simply by writing new "decorators".

These properties lead to this simple disk interface

```
save( value ) : reference
load( reference ) : value
```

This section covers how to locate (load) values saved on disk (section 5.2.1), which types of value may be stored on disk (section 5.2.2) and finally discusses some performance issues in section 5.2.4. The design and implementation of the disk API is described in chapter 6.

5.2.1 Locating values

When loading a value it is naturally necessary to locate it first. Locaters are used to locate values. Locators are, as described in chapter 3, a description of the exact route to a value. A local locator locates a value on a given disk, e.g. by offset and length. A local locator is not enough to locate values in a distributed environment (or even in an environment with multiple disks on the same computer) as they hold no information about the disk. A global locator is needed. A global locator is a local locator and a route to a disk e.g. ip, port, disk name. A protocol for locating values could look like the protocol in figure 5.5 (using *Universal Resource Identifier* (URI)[5, p.356] syntax). XML Store peers do not need any explicit knowledge of other peers because locators contain precise (and independent) route information. The information needed to retrieve a value is always found in the locator. Therefore a connection can be established to the peer where the value resides.

Only using locators has shortcomings. Such an architecture does not support mobility of values. If data is moved from one physical location to another

$$\underbrace{xmls : \underbrace{//ip : port/disk}_{route\ to\ disk} : \underbrace{offset : length}_{local\ locator}}_{global\ locator}$$

Figure 5.5: Simple Locator protocol

physical location the locator becomes obsolete. Thus mobility of values cannot be supported only using locators to reference values. A solution would be to put a forwarding locator at the values old position, showing the new position. However this could lead to long chains of locators, which is difficult to maintain and not desirable.

Besides not supporting mobility of data, problems occur when not actually writing values to the physical disk when the save operation is called (e.g. when implementing write-buffering or asynchronous-write). When write-buffering is considered, disks must buffer incoming values until actually writing a buffer of values becomes feasible. However, locators are (and can only be) created when values are actually saved as they need to know the actual location on disk. Since the save call should return a locator to the position of the value, not writing values right away becomes difficult.

As already described value references are used when values are saved, and these must first be looked up using the reference server to retrieve a locator. Resolving a value from a value reference is done using the reference server to retrieve a locator as described in section 5.1.

Value references are built from the content of values (as described in section 3.3.1). Write-buffering (or asynchronous-write) is now easy, just compute and return the value reference and write the value when feasible (e.g. the buffer is full). Value references also enable mobility of values to be implemented, since value references do not become invalid, if values are moved.

As looking up locators can be expensive, value references should cache looked up locators for future value retrievals. When the cached locator becomes obsolete (if the value is moved) a new must be looked up using the reference server.

Value reference offers other interesting possibilities, e.g. multiple locators could exist for each value reference, this would enable schemes for loading from peers located physically closest.

Computing value references

Value references are, as mentioned in section 3.3.1 content hashed values. Requirements to the hash function are also listed in section 3.3.1.

In the prototype implementation, MD5[33] is used. MD5 produces 128 bit digest. It is fast on 32-bit architectures. To find two messages that have the same digests takes 2^{64} operations. Finding a given message from a digest takes 2^{128} operations [33, p.280]. However, choosing other hash functions e.g. SHA1 is a simple implementation issue.

5.2.2 Storable values

So far, data that can be stored (and loaded) from disk has been called values.

Values are, as described in chapter 3, immutable objects. The disk must admit arbitrary sized values for input and output. This property implies that values should be loaded/saved lazily, which leads to the concept of streams (or infinite lists of values). Stream of values or value streams enable values to be read lazily. Value streams are compound values, consisting of either bytes (data) or references to other values (value references). As disks must admit arbitrarily sized input/output bytes are also returned in streams. Value streams can be expressed like:

```
valuestream = bytes (valuereference bytesequences)*
```

where bytesequences is defined as

```
bytesequences = byte*
```

This regular expression shows that value streams consist of bytes followed by a sequence of value references and bytes, which is repeated zero or more times. Bytesequences are just data, consisting of zero or more bytes.

Value streams

Values are loaded using value streams. A value stream is as defined above at stream (lazy list) of values, and is a value itself. That streams are values contradicts the conventional (imperative) perception of streams, where data are removed from the stream when retrieved. Value streams, however, act like lazy lists in Haskell, retrieving the first value from a given stream always returns the same value. Therefore, value streams must offer functionality to retrieve the first value from the stream and the rest of the stream. Consider figure 5.6 which illustrates a value stream "holding" the data "foobar". The value stream consists of a byte stream ("fo") and a reference to another value (that may be located on another disk on another machine), an empty byte stream (this is necessary in order to conform to the above grammar and a reference to yet another value.

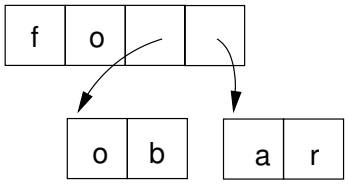


Figure 5.6: value streams and references

These properties of the data saved on disk enables disks to "natively" store tree-structured data, the following XML document could (with minor adjustments) correspond to figure 5.6

```
<fo>
  ob
  <ar/>
</fo>
```

The disk interfaces do not introduce the notion of tree-structured data, disks can also be used to persist non-tree-structured data.

5.2.3 Cells

As described in chapter 3 introducing updateable variables will extend the programming model and give application programmers more flexibility. Such variables are called cells. The disk offers a possibility to differentiate between references to values and references to cells by introducing *cell references* that support functionality to set (update) the value referred to. This means that the value reference from the above grammar, is substituted with a reference, that is:

```
reference = valuereference | cellreference
```

Cell references are designed such that they hold a reference to a value, the set operation is called with another value reference to change the value in the cell.

References to cells cannot be content-hashed as the content may change. Instead another scheme must be used, e.g. global unique id (guid), which may be a random sequence of bytes or (just) a locator.

5.2.4 Performance issues

This sections presents different strategies for improving performance, taking advantage of the value-oriented properties discussed in chapter 3.

Caching values can easily be implemented as values are immutable and coherence protocols are therefore not needed. Caching loaded values prevent these from being loaded multiple times. Caching values can be introduced different places in the system.

1. value streams can memorize bytes already read such that when retrieving the same bytes multiple times, they are only loaded from disk once.

2. value references can cache locators to prevent multiple lookups. When mobility of values is considered this would require some simple protocol for loading e.g. first using cached locators and then on fail (the value have been moved), lookup a new locator.

3. disks may cache loaded value streams in memory, such that when loading a value reference the cache is checked before any potential lookup is performed.

4. disk may cache (replicate) loaded values on disk, such that future retrievals does not require a remote load.

As mentioned above, such caching schemes can easily be implemented as values are immutable. Cells do not offer these possibilities. If these are to be cached traditional caching protocols must be implemented. The prototype implementation given with this thesis simply does not cache cells.

Considering that the disk layer is used by the DVM layer, saving a document may be an expensive process as each node must be looked up. To prevent large response times when saving documents, documents can be saved asynchronously, such that calling save returns a value reference without saving anything, but starts a background process that saves the value. This can also be implemented without concern to coherence.

Another strategy for improving performance would be to *in-line* value streams, so that a value reference is only created when value streams actually are larger than value reference (128 bit). Such a strategy would improve performance as fewer calls to the reference server have to be made. This improvement is naturally aimed at XML documents containing many, but small, nodes. The prototype implementation uses in-lining.

5.3 Name server

The name service functionality of the Document Value Model interface is solved by a *name server*.

The name server provides a distributed name service for creating and maintaining bindings of human readable names to value references. XML Store peers use the name server to associate names with documents and share this association with other XML Store peers.

The name server functionality is similar to the reference server functionality. It should optimally have the same properties as the reference server (see section 5.1). I.e. it should be buildt as a **decentralized** peer-2-peer solution with a **flat network hierarchy**. The peers in the decentral name server network should be **stateless**. Name to value reference bindings should be persisted.

Opposite the reference server, the name server is imperative, which poses problems in a decentral name server architecture.

A value reference cannot be updated to refer to a different value, as described in chapter 3. A specific value reference can be mapped to different locators on different reference server peers, as the locators all address the same value.

This is different for the name server. A name can only bind one value reference. This poses a consistency problem. When creating a new binding, the existence of the name must first be checked by performing a distributed lookup using IP Multicast (as in the reference server solution). As IP Multicast is not reliable the lookup may fail even if the name exists. The result is an inconsistent state where the name is associated with two different value references (on two different peers).

A second problem resides in the imperative rebind functionality, which the reference server does not have. Since the location of document names are not known, a rebind must be performed by an IP Multicast as when performing a lookup. Due to the missing reliability, rebind messages may not be delivered to all peers.

The update problems can be avoided by implementing the name service as a central name server.

5.3.1 Central solution

The name service functionality in XML Store is implemented as a central solution.

A central name server contains all names in the name space (and their associated value references) of the XML Store name service. XML Store peers look up, bind and rebind names using the central name server.

Network communication between XML Store peers and the central name server thus takes place each time name service functionality is invoked on peers.

A central name server unfortunately introduces a bottle neck and a "single point of failure" in XML Store. The simple solution simplifices the update problem (described above). Inconsistent states are avoided since bindings are not spread on several locations as in a decentral name server.

Furthermore, the name server is not used as often as the reference server. The bottle neck problem is thus non-existing for small XML Stores, i.e. XML Stores containing a small amount of peers, which makes the solution satisfying for our prototype model.

Chapter 6

XML Store implementation

This chapter describes an implemented prototype of XML Store. It by no means gives a detailed description of the code, but presents an overview of the implementation. Only the most relevant interfaces and classes and the most relevant methods and fields are presented. For the interested reader the full source code can be found in appendix C.

The prototype implementation of XML Store is *configurable* and *extensible*. These properties are achieved by two means:

1. The implementation provides the application with a flexibility of functionality, i.e. individual objects can be configured with specific values and functionality can be added without modifying and recompiling the existing code. An example given in section 5.2 shows how the decorator pattern [10, p.175] can be used, such that capability/functionality can be added to disks in compositional manner.

2. The implemented functionality is separated into *modules*. A module is a group of classes, providing well defined and related functionality. A module provides an interface, through which its functionality is accessed. Since functionality is enclosed in modules, implementations of functionality can be replaced with a different implementation of the functionality, by replacing the existing module with another module conforming to the same interface. This makes the prototype implementation configurable and flexible to changes.

   An example of a module is the reference server. This functionality is enclosed in a module, which conforms to the simple interface presented in 5.1. The actual implementation of the interface can be changed.

The layers presented in chapter 5, DVM layer and Disk layer, have a modularized structure. The DVM layer has two modules, a module supplying core DVM functionality and a module supplying name server functionality. The Disk layer has a similar structure as it has a module supplying core disk functionality and a module supplying reference server functionality.

This chapter describes implementations of each module. Section 6.1 and section 6.2 presents the core DVM functionality and the core disk functionality respectively. Section 6.3 presents the reference server module. The name server

module is not described in this chapter as the implementation, as mentioned in 5.3.1, is a simple insufficient central server solution.

6.1 DVM core module

The DVM core module provides the functionality of the Document Value Model (DVM) interface. To provide this functionality the module uses the name server and the core disk module. It is implemented in the Java package `edu.it.dvm`.

Among the java classes in the DVM core module `DVMXMLStore` is central. The class implements the `XMLStore` interface and is responsible for access to name service functionality and persistence functionality.

Figure 6.1 illustrates how the `DVMXMLStore` hold references to a `Disk` instance and a `NameServer` instance.

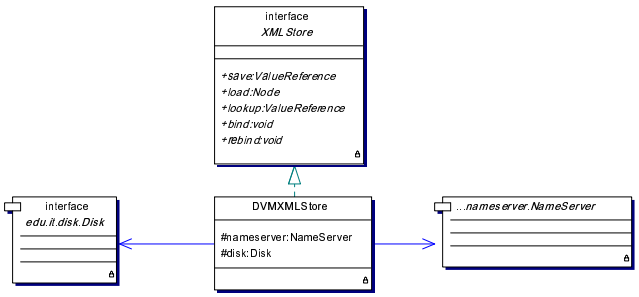


Figure 6.1: `DVMXMLStore`, the class implementing `XMLStore`.

XML documents in DVM are represented by `Node` and `MutNode` interfaces. The `Node` interface is implemented by `DVMImmNode` and `DVMImmNodeProxy`. The first represents nodes created and residing in memory while the second represents nodes on disk. `DVMMutNode` represents mutable nodes. Figure 6.2 show the classes and their relations.

6.1.1 Loading documents

XML Store must be able to handle documents of arbitrary size. A loading strategy where documents are loaded fully into memory is therefore not applicable. The applied loading strategy assures, that whole documents are not loaded into memory. Instead they are loaded lazily. I.e. document content is only loaded when being requested.

A lazy loading strategy for XML documents is implemented by loading immutable nodes lazily. The class `DVMImmNodeProxy` implementing the `Node` interface represents persisted immutable nodes. An instance of `DVMImmNodeProxy` is a proxy for a persisted node. The proxy loads persisted node data the first time this data is requested (through use of the node interface).

Proxies do initially not contain node data. Instead the data is loaded when being accessed. In order to load the node data, the proxy instance contains a value reference to the data, which is used to load the data.

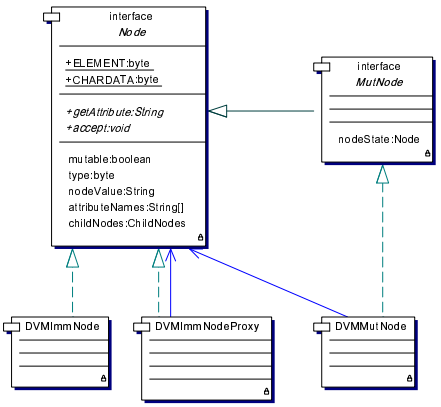


Figure 6.2: Classes implementing `Node` and `MutNode`.

Loading a node follows this procedure:

- Loading a node with a given value reference returns a proxy object containing the value reference.
- When a method is invoked on the proxy object, the requested node data is loaded.

The five methods `getType`, `getNodeValue`, `getAttribute`, `getAttributeNames` and `getChildNodes` loads data if not it is not already loaded.

That child nodes are loaded lazily, means that child nodes not accessed is not loaded from disk. Huge parts of a document may therefore not be loaded from disk. This is illustrated in figure 6.3.

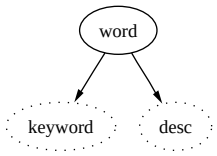
As more and more nodes are loaded and their data as well, memory usage increases. For huge XML documents this imposes a problem, as traversing such document eventually will fill up memory.

The strategy for minimizing memory usage is implemented by constructing a FIFO list holding node proxies. When node proxies loads either node value, attributes or child nodes, they register themselves in the list. If the queue is full, a node proxy is removed from the list and the data of the node proxy is discarded from memory.

When node proxies are removed from the queue and their data is discarded, they again only contain the value reference to the node data on a disk. Node data must thus be loaded again, if a proxy node has been removed from the queue.

Node proxies only register themselves in the queue the first time they load either node value, attributes or child nodes. When a node proxy is in the queue and more data is loaded, it does not register itself in the queue again. Keeping memory usage down by implementing a FIFO list is simple and straightforward.

A more sophisticated solution could be to implementation a strategy that consider how frequently data is requested. Data less frequently accessed is discarded from memory before more popular data.



The child elements of the “word” element is loaded. Only proxy objects are retrieved for the child elements.

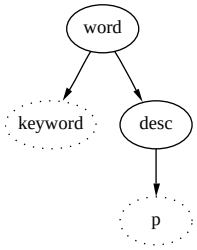


Figure 6.3: The “desc” elements child node is loaded. The dotted ellipses denotes proxy objects, whose children is not yet loaded.

Summery

Nodes are fully loaded (using the proxy approach descriped above) regardless of size, even when nodes represents a large amount of persisted data, all data are loaded. This may cause problems when:

1. Element nodes with a large number of children. All references are loaded and the same number of proxy objects are created.

2. Node values are large (most likely to happen for character data nodes). The whole value is loaded into memory. A stream based retrieval would solve this problem. (Such is implemented in the disk layer but not propagated to the DVM layer).

6.1.2 Saving documents

XML documents represented by the Document Value Model can be saved using the `XMLStore` interface. The Document Value Model allows sharing of nodes within an XML document and between XML documents (as described in chapter 3). This sharing must be kept intact when saving documents.

XML documents are saved in postorder traversal by saving one node at a time.

Nodes are saved by using the disk module. This requires unparsing nodes into value streams (defined in section 5.2.2), which are then written to disk. How nodes are unparsed depends on the node types. When a node represents

1. character data, the corresponding value stream contains the character data as a byte sequence.

2. an element, the corresponding value stream consists of a preceding byte sequence followed by zero or more value references separated by empty byte sequences (in order to follow the value stream definition in section 5.2.2). The preceding byte sequence contains the element name and the attributes. The value references are references to the child elements.

Postorder traversal saving means that child nodes are visited before parent nodes. This is necessary in order to produce the value references for the children. If nodes in a document, i.e. children of an element, have already been saved and are represented by either `DVMImmNodeProxy` instances or `DVMMutNode` instances they are not unparsed and saved again. Instead their already existing value references / cell references are retrieved (from the `DVMImmNodeProxy`/`DVMMutNode` instances) and used when saving the relevant element.

Saving documents is implemented in class `SaveVisitor`.

Figure 6.4 illustrates how an XML document is transformed into value streams.

Saving documents by saving one node at a time makes it possible to support sharing of arbitrary nodes (and subtrees) within and between documents persisted on disk.

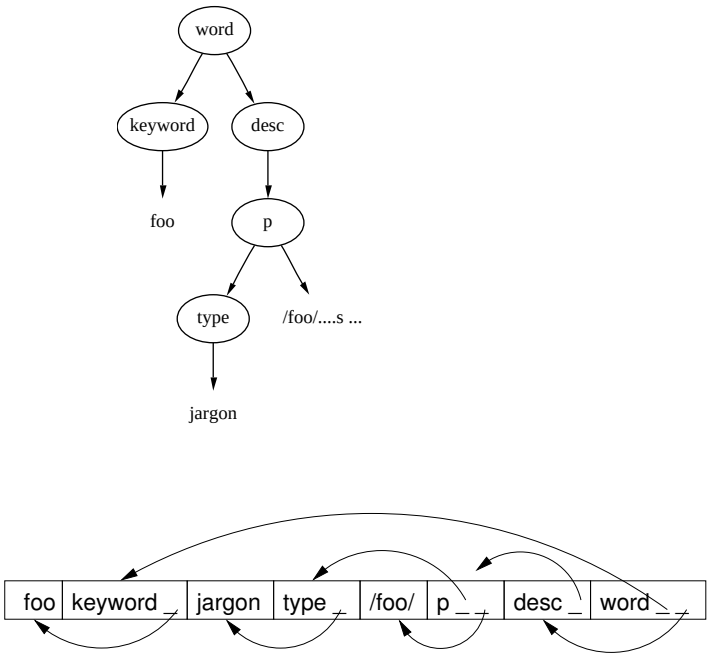


Figure 6.4: Unparsing nodes into value streams. Each node is unparsed and saved to disk. The bottom figure illustrates the resulting sequence of value streams (as they are saved to disk storage) when unparsing the XML document from the top figure. '|' separates valuestreams, '_' combined with arrows indicate references. The figure is a simplification made for illustration purposes.

6.1.3 Mutable nodes

Mutable nodes are represented by the class `DVMMutNode`, which implements the `MutNode` interface. Both mutable nodes, which have not been saved, and mutable nodes, which have been saved, are represented by `DVMMutNode`.

Mutable nodes which have not been saved resides solely in memory. When invoking the `getNodeState`/`setNodeState` methods on such nodes, the node state kept in memory is returned/set.

If the mutable node has been saved, `getNodeState` and `setNodeState` do not just return/change the node state kept in memory. The methods return/change the node state persisted on disk.

When `DVMMutNode` instances have been saved, they do not only contain a value reference like `DVMImmNodeProxy` instances do. They contain cell references. Changing a persisted node's state is therefore done by saving the new node state and set the cell reference to hold the value reference returned from saving.

That the node state of persisted mutable nodes is loaded and saved each time `getNodeState` respectively `setNodeState` is called increases the probability, that the node state seen by the application program is consistent with the persisted node state. No guarantees exist though, as this would require transaction control due to the imperative nature of mutable nodes.

6.1.4 Child nodes

The data structure keeping child nodes is called *childlist*. The choice of data structure affects access and insertion times of Nodes.

Child lists are represented as arrays. Each node representing an element has an array of child nodes. The arrays are encapsulated in the interface `ChildNodes`.

Using arrays provides constant time for accessing an arbitrary node in the array. Thereby arbitrary children of a node can be accessed in constant.

However, inserting an element into an array representing a child list is expensive. As child lists are value-oriented, inserting an element means creating a new array by copying references to all existing elements, hence the add operations takes $O(n)$.

Perhaps a binary search tree would be better suited for representing child lists, as descriped in Baumann et al. [9, p.51]. Such a solution would give insert times of $O(log\ n)$, however, accessing an index in this representation also takes $O(log\ n)$, not $O(1)$ as the array solution.

6.2 Core disk module

The core disk module is the lowest level in the XML Store. The core disk module provides functionality for writing data on the physical disk and providing access to the data. As described in chapter 5, values can be stored and retrieved from disk using value streams. Values that can be stored are either bytes or references to values. The core disk module is implemented in the java package `edu.it.disk`. The disk module uses the reference server module to lookup locators.

Section 6.2.1 describes the API offered by the disk layers, i.e. which basic operations the disk layer offers. Section 6.2.2 describes how the log-structured filesystem is implemented and how values are loaded and saved on the physical disk. Section 6.2.3 describes how value streams are implemented.

6.2.1 Disk Interface

This section describes the basic functionality offered by the disk layer. The API described is used by the DVM module to implement a tree structured API, as described in section 6.1.

Creating value streams

Value streams can be obtained by loading references from disk, as we have just shown. To fill disks with new data it must be possible to create value streams from scratch. Four basic operations for creating value streams are offered. These are:

```
ValueStream mkEmpty( );
ValueStream mkByteValue Stream( byte[] first, ValueStream rest );
ValueStream mkValRefValue Stream( ValueReference first, ValueStream rest );
ValueStream mkCellRefValue Stream( CellReference first, ValueStream rest );
```

Building a value stream equal to the one illustrated in figure 5.6 looks like

```
Value Stream ob = mkByteValueStream(new byte[]{'o', 'b'}, mkEmpty());
Value Stream ar = mkByteValueStream(new byte[]{'a', 'r'}, mkEmpty());

ValueReference obref = save(ob);
ValueReference arref = save(ar);

ValueStream vs = mkByteValueStream(new byte[]{'f', 'o'},
                   mkValRefValueStream(obref,
                     mkByteValueStream(new byte[]{}, arref)
                   )
                 );
```

As value streams may be streams to disks, it is possible to create value streams that include some value in-memory, some on disk and some on other computers.

Using value streams

Value stream offer a simple API for retrieving values from streams. Value streams are values themselves and act different from "traditional" imperative streams, where data is removed from the stream on retrieval. Retrieving a value from a value stream will not remove this value from the stream, to obtain the rest of the stream a separate method must be called or the get method must return a pair consisting of the value and the rest of the stream, this leads to these simple interfaces:

```
public interface Value Stream{
   ValuePair getNext();
}

public interface ValuePair{
  Value first();
  Value Stream rest();
}
```

This has a number of pragmatic disadvantages:

1. as it is necessary to differentiate between values returned form `first()` which may be either bytes, a value reference or a cell reference (as expressed in the regular expression for value streams), an expensive and unsafe downcast is required to get the proper type from `Value`.

2. values may be very large, the interface does not offer any means of loading values lazily. I.e. using the method first the whole value is loaded.

The solution to 1) is to introduce type specific methods for retrieving bytes, value reference and cells and a method to get the type of the next value. Solution to 2) is to offer bulk read of bytes. This produces the following value stream interface

```
public interface ValueStream{
  /**
   * @return the type of the first value in stream.
   */
```

```
  byte getValueType( )

  /**
   * @throws Exception if(getType() != BYTES)
   * @return all bytes until next valuereference
   *      (Possibly large, use method with care)
   * and remainder of valuetream
   */
  ByteArrayPair getBytes( ) throws DiskException;

  /**
   * @throws Exception, if(getType() != BYTES)
   * @return max num bytes and the remainer
   *      of the value stream
   */
  ByteArrayPair getBytes( int num ) throws DiskException;

  /**
   * @throws Exception,
   *      if( !(getType() == VAL_REF || getType() == CELL))
   * @return ValueReference and remainer of stream
   */
  ValueReferencePair getValueReference( ) throws DiskException;

  /**
   * @throw Exception, if(getType() != CELL)
   * @return Cell and remainer of stream
   */
  CellReferencePair getCell() throws DiskException;

  /**
   * throw away first value either valuereference or all bytes
   * @return rest of value streamf
   */
  ValueStream skip( )
}
```

The different Pair types all offer methods `CorrectType first()` and `Value Stream rest()`. Where `CorrectType` is either `ByteArray`, `ValueReference` or `CellReference`.

Method `skip` skips the next value. With this interface retrieving (and printing) all bytes from a value stream can be done like this:

```
public void printVS( Value Stream vs ){
  switch( vs.getType() ){
    case EMPTY:
      /* stop recursion */
      break;
    case BYTES:
      ByteArrayPair pair = vs.getBytes( );
      System.out.println( new String( pair.first() );
      printVS( pair.rest() );
      break;
    case VAL_REF: case CELL:
```

```
      ValueReferencePair pair = getValue StreamReference();
      Value StreamReference vsr = pair.first();
      printVS( vsr.get() ); // or printVS( disk.load(vsr) );
      printVS( pair.rest() );
  }
}
```

As the printVS method does not modify cells it is not necessary to differentiate between cells and value references. This is possible as cell references are subtypes of value references. To get a value stream referred to by a value reference, simply call method `get` from the value reference.

6.2.2 Log-structured file system

As mentioned in chapter 5 values are in a saved log structured fashion [35] on disk. To represent a file system we use Java's `RandomAccesFile`. `RandomAccessFile` provides access to files. A cursor called the *file pointer* indicates the current position in the file. When reading or writing, the file pointer is incremented. The file pointer can be moved to another position before each read or write procedure. These properties makes `RandomAccessFile` suitable for simulating log structured files.

Saving

Saving a value to the log structured file system is easy, simply save it at the end of the log. This is easily implemented by always keeping a file pointer a the end of the file. This way writing becomes efficient as no time is spent seeking a place to write.

Loading

Loading values is done using a locator. Locators hold offset and length of values to be loaded. Loading a value is done by opening a stream for retrieving the value. The stream starts at the specified offset and ends at the offset plus the specified length.

In a distributed environment locators also hold ip number and port number of the machine on which the value is located. Loading remote values is done similar to loading local values, a stream to the remote machine is opened and values are retrieved from here.

6.2.3 Streams

The streams used to access the underlying random access file are, as described in section 5.2.2, value streams.

Value streams use memorization of values to prevent unnecessary disk traffic. Memorization in this context means that streams cache values read, such that when retrieving a value a second time, it is not loaded from disk. Memorization can easily be implemented as values are immutable and no coherence protocols are needed.

In the prototype three different concrete value streams are used, their relationship is depicted in figure 6.5. The class `SimpleValueStream` is used

when streams are created from values residing in memory, e.g. when nodes are saved they are "flattened" into value stream (as described in section 6.1.2). `SimpleValueStream`s are created using the create methods in the `ValueStreamFactory`. The class `DiskValueStream` reads from the random access file and the `GlobalValueStream` reads values from other XML Store peers in the system.

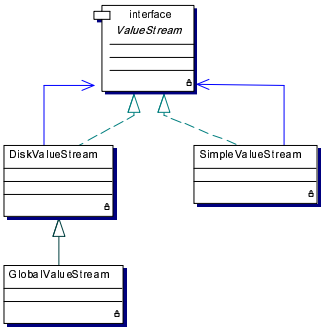


Figure 6.5: Class diagram of value streams.

`DiskValueStream` and `GlobalValueStream` are built on top of regular imperative streams. To implement these, two classes that extend Java's `InputStream`[1] class are implemented, a `RAFInputStream` to provide access to a `RandomAccessFile` and a `SocketInputStream` to provide access to another machine. Extending Java's `InputStream` has the advantage that already implemented functionality such as `BufferedInputStream`[2] can be used easily. Figure 6.6 shows a class diagram of the imperative input streams used to build value streams.

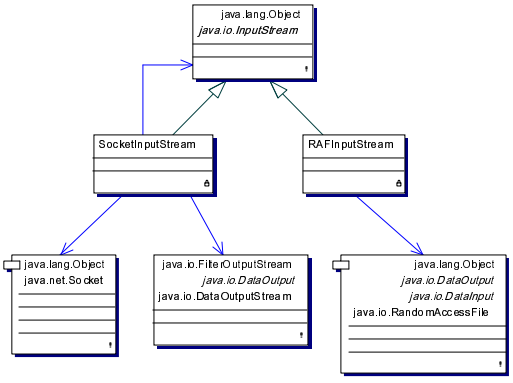


Figure 6.6: Class diagram of imperative streams

[1] InputStream represents an input stream of bytes. The class interface is imperative.
[2] `BuffereredInputStream` adds buffer functionality to another input stream.

Loading values from a local disk is done using a `DiskValueStream` that use a `RAFInputStream` to load values from the `RandomAccessFile`. The `RAFInputStream` is given an offset and a length (supplided by a locator), and data between offset and offset + length can be read using the `RAFInputStream`.

Loading values from a global disk is done using a `GlobalValueStream` that use a `SocketInputStream`. The `SocketInputStream` is given an ip, port, disk name, offset and length (supplied by a global locator). The ip, port and diskname specifies the location of the disk where the value is located. The offset and length specifies the location of the value on the disk, as explained in section 5.2.1. The `SocketInputStream` makes a socket connection to the disk specified by ip and port. When data is retrieved through the `SocketInputStream` (when it's `read` method is called), it sends a request to the disk (via a socket) to read (a certain amount of) data. The disk reads the data using and `RAFInputStream` and sends it back though the network.

6.3 Reference server module

The functionality provided by the reference server module is a distributed service for mapping value references to locators, as described in section 5.1. The reference server is the reference resolver discussed in section 3.3.1.

The java package `edu.it.disk.refserver` contains the reference server module, which has the interface represented by `ReferenceServer`. The `ReferenceServer` interface provides two methods `lookup` and `bind`.

The class `GlobalReferenceServer` implements `ReferenceServer`. `GlobalReferenceServer` instances are responsible for performing the reference service protocol described in section 5.1.1.

A `GlobalReferenceServer` instance contains a reference to a `LocalReferenceServer` instance. `LocalReferenceServer` acts as a local persistent hashtable containing value reference to locator bindings.

`GlobalReferenceServer` instances use a `Communicator` to perform all distributed communication. The `Communicator` interface contains two methods, `send` and `sendReceive`, which are defined below. The class `MulticastCommunicator` implements `Communicator`. `GlobalReferenceServer` actually contain a `MulticastCommunicator` instance. `MulticastCommunicator` instances send messages to and receive messages from other `MulticastCommunicator` instances. The distributed communication in the reference server is fully encapsulated in `MulticastCommunicator`.

Figure 6.7 illustrates the interfaces and classes and their relations.

Classes implementing `Comunicator` must implement two methods:

`void send(byte[] msg)`
: sends *msg* through the network. In case of the `MulticastCommunicator` *msg* is sent to all subscribers of a specificed IP Multicast group, as described in section 5.1.

`int sendReceive(byte[] msg, byte[] response )`
: sends *msg* through the networks, waits for response (which is put into *response*) and returns the number of bytes in response. The `MulticastCommunicator` sends the msg to all subscribers of a specifed IP Multicast

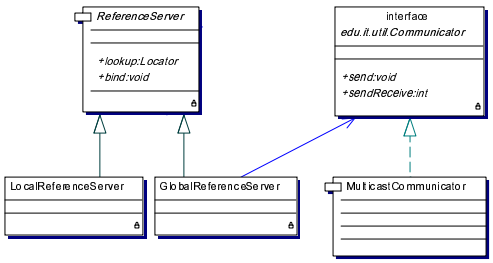


Figure 6.7: Class diagram of reference server

group and wait for response from any subscriber, when a response is received all other responses are ignored.

The reference service protocol implemented in `GlobalReferenceServer`, requires a distributed lookup. This is done using the `sendReceive` method of the `Communicator` interface. The lookup request package has the following structure:

```
ACTION | LENGTH | value reference
```

where `ACTION` is a byte, which denotes the requested functionality and `LENGTH` is the length of the key to lookup, although unnessesary (as value references has a fixed size) it is added to make the protocol general.

In turn the response just includes the locator. The `Communicator` handles protocol information about which port the response should be sent to and a request id ensuring that requests get correct responses.

Bind are made only using the `LocalReferenceServer` since distribution of mappings are not performed (as described in section 5.1).

Chapter 7

Evaluation of the Document Value Model.

The Document Value Model (DVM) is the API of the implemented XML Store prototype. This API should be usable and adequate for working with XML documents. The value-oriented programming model should provide advantages compared to the imperative programmring model when building distributed XML applications, as specified in chapter 1.5 and chapter 3

In order to compare the value-oriented programming model with the imperative programming model, DVM is compared to the Document Object Model (DOM). We compare these two approaches by implementing a sample application.

To give an introduction to programming with DVM, a small application that counts nodes in an XML document is presented in section 7.1. This section provides a quick tour of DVM and illustrates some of the advantages of DVM. Section 7.2 presents the dictionary application. DOM version is presented in section 7.2.1, the DVM version is presented in section 7.2.2. Section 7.2.3 compares the two different implementations and states differences in the two programming models.

7.1 Node counter

To introduce programming with DVM, we present a simple application counting nodes (i.e. Elements and Character Data) in an XML document.

The node counter application is implemented as a single Java class, `NodeCounter`. The class contains the method `public static int count(Node node)`. Given a DVM Node (see section 4.1) the method recursively counts the number of nodes in the sub tree represented by the node.

The `NodeCounter` class is shown below (exception handling and imports of necessary Java packages are not included).

```
1  public class NodeCounter{
2     public static int count( Node node ){
3        int nodes = 1;
4        ChildNodes children = node.getChildNodes();
5        for( int i = 0; i < children.getLength(); i++){
6           Node child = children.getNode(i);
```

```
7            if(child.getType() == Node.CHARDATA)
8               nodes++;
9            else
10              nodes += count( child );
11       }
12       return nodes;
13    }
14
15    public static void main(String[] args){
16       String storeName = args[0];
17       String docName = args[1];
18       XMLStoreFactory factory = XMLStoreFactory.getInstance();
19       XMLStore xmlstore = factory.createXMLStore(storeName);
20       ValueReference ref = xmlstore.lookup(docName);
21       Node root = xmlstore.load(ref);
22       int nodes = count(root);
23
24       System.out.println("Document node count : " + nodes);
25    }
26 }
```

The main method which should be invoked with two arguments, xmlstorename and name of document (bound on nameserver), initializes an XMLStore, calls the count method and prints the result.

The application can be invoked from the command line through the call:

```
java NodeCounter myxmlstore mydoc
```

This example illustrates how a simple application, which count nodes, easily can be implemented using the XML Store.

When counting nodes in an XML document the document structure is traversed. The `count` method illustrates (in lines 4-6) how documents are traversed.

The example illustrates that lazy loading is done transparently. Lazy loading has the effect that character data nodes are never loaded. Line 7 accesses the type of a node, calling `getType()` only the byte representing node type, hence the node content is never loaed, as only element nodes are processed further. In contrast to DOM only information needed is loaded from disk.

Furtermore the example illustrates that location of nodes, i.e. in memory, on disk or on another peer, is not an issue. No code in the example (explicitly) loads nodes into memory (from disk or network). The `NodeCounter` class can be used to count nodes from any document regardless of location.

Location transparency is further illustrated in the main method. The document name (`docName`) contains no information of location. Retrieving the root node (lines 20-21) reveals no information about location.

7.2 Dictionary

A dictionary is a

> *reference book containing an alphabetical list of words, with information given for each word, usually including meaning, pronunciation and etymology* [36].

I.e. dictionaries are used to provide people with a common understanding of words by providing information for each word.

The *dictionary application* is used to search words in a dictionary and insert new words (and their definition) into the dictionary. The application maintains the dictionary data in an XML document called the *dictionary document*. The dictionary document is accessible from any location and the data can be accessed by several applications.

The dictionary application should be buildt, such that several users can start the application and use the same dictionary document. The location in which the application is started should not matter.

Examples given throughout the report is taken from or influenced by the dictionary application. The structure of the dictionary document is presented in appendix B. Words searched for or inserted into the dictionary are called *keywords*. As also stated in appendix B, word elements in the dictionary document are sorted according to their keyword's lexicographical order.

The dictionary application itself has the following properties:

- searches are performed binary by comparing keywords.
- successful searches returns the word element, corresponding to the searched keyword.
- unsuccessful searches returns

  `<word><keyword>No match on keyword</keyword></word>`.
- inserts do not violate the alphabetical order of the dictionary.
- it is not possible to insert a word already defined in the dictionary document.

To compare the value-oriented programming model to the imperative programming model, we have made implementations using both DOM and DVM.

Section 7.2.1 presents the dictionary implemented using the DOM interface. Section 7.2.2 presents the dictionary application implemented using XML Store. Section 7.2.3 compares the two different implementations.

7.2.1 DOMDictionary - an imperative dictionary application

DOMDictionary is a dictionary application implemented using the Document Object Model interface. The dictionary document is a serialized XML document stored in a flat file.

The application is implemented as a single Java class, `DOMDictionary`.

The class contains a constructor, three private methods `load`, `save` and `binarySearch` and the public methods `keywordSearch` and `insert`.

The constructor takes one argument, which is the file name (inclusive path) of the dictionary document.

Methods `loadDict` and `saveDict` are used to save and load a dictionary document. `loadDict` parses the dictionary document to a DOM representation in memory. `saveDict` unparses the DOM representation in memory and saves (overwrites) in the flat file containing the dictionary document.

The private method `binarySearch` is used by the methods `keywordSearch` and `insert` to search for keywords in the dictionary document. The method

performs binary search on dictionary keywords. If the keyword is found, the method returns true and the private class variable `match` is updated with the word element containing the keyword. Otherwise the method returns false.

`keywordSearch` is used to perform a keyword search. The method first loads the dictionary document to retrieve the latest version of the document. Thereafter it invokes `binarySearch` to search the keyword. If the keyword exists, the result in `match` is returned. Otherwise a DOM document containing the message `'No match on keyword'` is returned.

The method `insert` is used to insert new keywords and their definition into the dictionary document. The method takes as argument a DOM node representing the word element with the new keyword and its definition. First the dictionary document is loaded in order to retrieve the latest version of the document. Thereafter the new keyword is searched in the dictionary. If it already exists an exception is thrown. If not, the new keyword is inserted and the dictionary document is unparsed and written to storage.

The Java package `javax.xml.transform` and its subpackages are used to unparse XML documents in DOM representation to a serialized representation and write to output destination. This is done in lines 98-102 and lines 114-118 of the Java code seen below. Imports of Java packages and exception handling are omitted to simplify the code.

```
1  public class DOMDictionary{
2    private Document dict;
3    private DocumentBuilder builder;
4    private Transformer transformer;
5    private Node match;
6    private String dictName;
7
8
9    public DOMDictionary( String dictName ){
10     this.dictName = dictName;
11     match = null;
12     System.setProperty(
13        "javax.xml.parsers.DocumentBuilderFactory",
14        "org.apache.xerces.jaxp.DocumentBuilderFactoryImpl");
15     DocumentBuilderFactory factory =
16        DocumentBuilderFactory.newInstance();
17     factory.setIgnoringElementContentWhitespace(true);
18     builder = factory.newDocumentBuilder();
19
20     TransformerFactory tFactory =
21       TransformerFactory.newInstance( );
22     transformer = tFactory.newTransformer();
23     transformer.setOutputProperty(OutputKeys.ENCODING,
24                                   "iso-8859-1");
25   }
26
27   private boolean binarySearch( String word ){
28     NodeList hws = dict.getElementsByTagName( "keyword" );
29
30     int a = 0, b = hws.getLength() - 1;
31     while( a <= b ){
32       int i = (a+b) / 2;
33       Node n = hws.item( i ).getFirstChild();
34       String s = n.getNodeValue();
35       int res = s.compareToIgnoreCase( word );
36       if ( res > 0 ){ // greater, search right
37         match = hws.item( i ).getParentNode();
38         b = i - 1;
```

```
39        }else if ( res < 0 ){ // smaller, search left
40          match = hws.item( i ).getParentNode();
41          a = i + 1;
42        }else{ // found!
43          match = hws.item( i ).getParentNode();
44          return true;
45        }
46      }
47      return false;
48    }
49
50    public Node keywordSearch( String keyword ){
51      loadDict();
52      Document doc = builder.newDocument();
53      if( binarySearch(keyword) ){
54        Node n = doc.importNode( match, true );
55        doc.appendChild(n);
56        return doc;
57      }
58      Node res = doc.createElement("word");
59      Node kw = doc.createElement("keyword");
60      doc.appendChild(res);
61      res.appendChild(kw);
62      kw.appendChild(doc.createTextNode("No match on keyword "
63                                        + keyword));
64      return doc;
65    }
66
67    public void insert( Node word ){
68      NodeList nws =
69        ((Element)word).getElementsByTagName( "keyword" );
70      // only one keyword element exist within word
71      Node child = nws.item(0).getFirstChild();
72      String keyword = child.getNodeValue();
73
74      loadDict();
75      if( binarySearch(keyword) ){
76        //allready exists throw exception
77        throw new DictionaryException("Word exists!,
78                                      "nothing inserted");
79      }
80      Node parent = match.getParentNode();
81      Node newNode = dict.importNode(word, true);
82      parent.insertBefore(newNode, match);
83
84      saveDict();
85    }
86
87    private void loadDict(){
88      // concurrency control needed
89      // but not implemented
90      dict = builder.parse( new File( dictName ) ) ;
91    }
92
93    private void saveDict(){
94      /* concurrency control needed when saving but not
95         implemented, i.e. if Dictionary has been modified
96         by other processes saving should be aborted new
97         document loaded, and insertion tried again */
98      BufferedWriter bwrt =
99        new BufferedWriter(new FileWriter(dictName));
100     transformer.transform(new DOMSource(dict),
```

```
101                             new StreamResult(bwrt));
102       bwrt.close();
103     }
104
105     public static void main(String[] args){
106       if( args.length < 2 ){
107         System.err.println(
108            "Usage: java DOMDictionary <dictionary file>" +
109            "<keyword>");
110         System.exit(-1);
111       }
112       DOMDictionary dict = new DOMDictionary( args[0] );
113       Node doc = dict.keywordSearch( args[1] );
114       Source source = new DOMSource(doc);
115       TransformerFactory tFactory =
116         TransformerFactory.newInstance();
117       Transformer transformer = tFactory.newTransformer( );
118       transformer.transform(source, new StreamResult(System.out));
119     }
120   }
```

The application is invoked from the command line with two arguments. The first argument is the file name of the dictionary document. The second argument is the keyword to search for. In the example belowm the dictionary document resides in the directory with the relative path `dict` and has file name `foldoc.xml` The word searched for is `foo`.

```
java DOMDictionary dict/foldoc.xml foo
```

7.2.2 XMLStoreDictionary - a value-oriented dictionary application

To evaluate the value-oriented programmering model we present an implementation of the dictionary application using XML Store. XMLStoreDictionary is a dictionary application implemented using XML Store for searching keywords and inserting new keywords into a dictionary document, i.e. the dictionary document is stored in the XML Store and is implemented using DVM.

Using an XML Store makes it possible to share the dictionary document between several processes. These processes will all be able to insert new keywords into the dictionary document. Between two searches in an XMLStoreDictionary application instance, the document can be updated by another process. This requires the document to be loaded for each search or insert, if it has been changed.

The application is implemented as the single class `XMLStoreDictionary`. The constructor initializes `XMLStoreDictionary` with an `XMLStore` instance and loads a value reference for the dictionary document.

The `binarySearch` method provides functionality equal to `binarySearch` in the DOMDictionary application. It is used both to search for keywords in the dictionary document and to find a position in the dictionary document where a new keyword should be inserted. The method is given a keyword as parameter. If the keyword is found, the method returns a non-negative integer. The keyword is contained within a word element. The returned integer denotes the word elements child index within the dictionary element (recall the dictionary documents structure described in chapter 2). E.g. if a small dictionary document contains 3 word elements sorted lexicographically by their

keywords, respectively “bar”, “foe” and “foo”, a search on “bar” will return the non-negative integer 0.

If a keyword is not found, the method returns a negative integer. This integer is used, if inserting a word element with the keyword into the dictionary document. The integer implicitly denotes the position in the dictionary element where the new word element must be inserted. The position is calculated by inverting the negative integer and subtracting one. That is if the above dictionary document example is used and “ccc” is searched the returned value is -2. The position is then calculated as: $-2-1=1$. After inserting “ccc” the dictionary contains 4 word elements still sorted lexicographically by their keywords, respectively “bar”, “ccc”, “foe” and “foo”.

The method `keywordSearch` searches for the keyword given in the method parameter. The method first loads the dictionary document if it has been changed by another process. Thereafter a binary search is performed and if the keyword is found, the word element containing the keyword is returned.

The method `insert` inserts a word element node into the dictionary. If the keyword within the word element already exists in the dictionary document, nothing is inserted. Otherwise the word element is inserted and the modified dictionary document is stored.

The XMLStoreDictionary example shown below is stripped from imports of Java packages, exception handling and standard in-/output code in order to simplify the example.

```
1  public class XMLStoreDictionary{
2    private XMLStoreFactory factory;
3    private XMLStore xmlstore;
4    private Node dict = null;
5    private ChildNodes words = null;
6    private ValueReference ref = null;
7    private DocumentBuilder builder = null;
8
9    public XMLStoreDictionary (XMLStore xmlstore){
10     this.xmlstore = xmlstore;
11     this.factory = XMLStoreFactory.getInstance();
12     this.ref = xmlstore.lookup("dictionary");
13     dict = xmlstore.load(ref);
14   }
15
16   private int binaryLookup(String word){
17     words = dict.getChildNodes();
18     int lo = 0;
19     int hi = words.getLength()-1;
20     int mid = 0;
21
22     while(lo <= hi) {
23       mid = (lo+hi)/2;
24       int compare = wordAtIndex(mid).compareToIgnoreCase(word);
25       if(compare == 0) return mid;
26       if(compare > 0) hi = mid -1;
27       else lo = mid+1;
28     }
29     return (-lo-1);
30   }
31
32   private String wordAtIndex(int mid){
33     Node keywordNode =
```

```
34        words.getNode(mid).getChildNodes().getNode(0);
35      return
36        keywordNode.getChildNodes().getNode(0).getNodeValue();
37    }
38
39    public Node keywordSearch(String keyword){
40      ValueReference ref = xmlstore.lookup("dictionary");
41      if(!this.ref.equals(ref)){
42        dict = xmlstore.load(ref);
43      }
44      Node word;
45      int index = binaryLookup(keyword);
46      if(index >= 0){
47        word = dict.getChildNodes().getNode(index);
48      }else{
49        Node kw = DVMUtil.createElement("keyword",
50                                        "No match on keyword " +
51                                        keyword);
52        word = factory.createElementNode("word", kw);
53      }
54      return word;
55    }
56
57    public void insert(Node word)throws DictionaryException{
58      ValueReference ref = xmlstore.lookup("dictionary");
59      if(!this.ref.equals(ref)){
60        dict = xmlstore.load(ref);
61      }
62
63      Node kw = word.getChildNodes().getNode(0);
64      String keyword = kw.getChildNodes().getNode(0).getNodeValue();
65
66      int index = binaryLookup(keyword);
67      if(index >= 0) {
68       throw new DictionaryException("Word exists!, " +
69                                     "nothing inserted");
70      }
71
72      index = -index-1;
73      dict = DVMUtil.insertChild(dict, index, word);
74      ref = xmlstore.save(dict);
75      xmlstore.rebind("dictionary", ref);
76    }
77
78    public static void main(String[] args){
79      String storeName = "dictionary";
80      XMLStoreFactory factory = XMLStoreFactory.getInstance();
81      XMLStore xmlstore = factory.createXMLStore(storeName);
82      XMLStoreDictionary dict = new XMLStoreDictionary(xmlstore);
83
84      do{
85        System.out.println("[Search 1, Insert 2, Exit 0]");
86        System.out.println("----------------------------");
87        ... // code performing search or insert depending on
88        ... // the above choice
89        ...
90        }while( true );
91    }
92  }
```

The XMLStoreDictionary application assumes that a dictionary document already exist and that the document name is `dictionary` (line 79).

7.2.3 Comparing DOM and DVM

The two dictionary applications are compared through different apects, all important when building (distributed) applications.

Convenience

Both APIs offer a tree-structed view of XML documents, giving them a high level interface when working with such documents as an abstract datatype. DOM is a larger API than DVM and offers a vast number of convenient operations, examples of these are `getElementByTagName` and `getFirstChild`. Such functionality must be coded by application programmers using DVM. However, the vastness of DOM may be confusing and difficult to learn.

Overall both APIs provide similar structure and functionality. This is illustrated in the example code. The DOM and DVM versions of the dictionary application are similar.

Memory consumption

DOM DOMDictionary reads the dictionary XML documents into memory. The dictionary document has a size of 7.6Mb, when stored in a flat file. This slows the DOM implementation, as 7.6Mb has to be read into memory at each search. Furthermore the dictionary may grow (when new words are inserted) to be to large to be loaded into RAM.

A solution could be, as mentioned in section 2.2.1, to implement indexes by separating the dictionary document into several files, e.g. a file for each letter in the alphabet. When searching and inserting keywords, the correct file is first loaded. The following example illustrates the `loadDict` method, when the file names are 'a_*dictName*', 'b_*dictName*', ..., 'z_*dictName*'.

```
87        private void loadDict(){
88          String fileName = keyword.substring(0,1) +
89          "_" + dictName;
90          dict = builder.parse( new File( dictName ) ) ;
91        }
```

Implementing such a scheme, requires application programmers to focus on external elements.

DVM XMLStoreDictionary is able to handle dictionary document files of any size, as the XMLStore does not load whole documents into memory.

Furthermore, only data needed is loaded into memory. Words that are never searched for is never loaded into memory.

Sharing of Nodes

DOM Data cannot be shared within or between XML documents. This means that every node belongs to a document. Using nodes from other documents (or even nodes within the same document), the given node has to be cloned. The `importNode` method in lines 54 and 81 illustrates how nodes are imported (cloned) into a document, and from then on belongs to the document. This leads to redundant data.

That nodes cannot be shared also means that DOM nodes have one and only one parent node. This means that method `getParentNode()` exists,

which may be helpful in certain situations, as illustrated at line 37, 39, 43 and 80.

DVM Data can be shared within documents and between documents. This is illustrated in line 73 where a new word is inserted by creating a new dictionary root node. The children of the old root node is shared with the new root node. When saving the new root node only the inserted child and the root node itself is saved. This is a clear advantage since expensive copying and ineffective saving of the unmodified data is not necessary.

Sharing nodes means, as discussed in section 3.2, that nodes have more than one parent (XML documents are dags, not trees), why it does not make sence to offer a `getParentNode` node operation. This means that application programmers must remember parent nodes if nessesary.

Sharing of nodes within the dictionary document exist. This is however most likely limited to sharing of small nodes, such as `<link>...</link>`.

Location transparency

DOM DOMDictionary does not have location transparency. As the XML document file has to be parsed into memory and written back to disk when new words are inserted, its position on the filesystem has to be stated. This is done with a command line argument, line 112.

Furthermore, application programmers are aware (or should be) that the dictionary document is loaded into RAM, and any change must be persisted back to disk. Line 90 parses the dictionary into an internal DOM tree, line 98-102 writes the updated dictionary to its location on disk.

DVM XMLStoreDictionary have location transparency. Even though the dictionary is loaded at Line 13, 42 and 60 and saved at line 74, no specific path in a filesystem is stated. Where the dictionary resides is transparent.

Furthermore application programmers do not have to differentiate between nodes in RAM and nodes on disk. They are treated equally.

Distribution transparency

DOM Distribution transparency is not provided by the DOM implementation. It may be provided to some extent by the underlying file system, such that actual location of data (files) is not revealed. E.g. the path `/import/stud/home/kasperp/dictionary.xml` in a Unix file system may not reveal, which machine the file is located at. It may be located at a local machine or at a network file system (NFS) server.

If distribution is not handled by the underlying filesystem (e.g. NFS), application programmers must write code for distributed saving and loading of files. The DOMDictionary application is therefore only capable of accessing dictionary documents saved locally or on mounted directories acting as local ones.

DVM The XMLStoreDictionary is distribution transparent. The dictionary may reside on remote XML Store peers. Lines 12-13 show how the dictionary is loaded without revealing where in the network it resides.

Parts of the dictionary document may reside on other XML Store peers, however the code remains the same. The DVMDictionary code works with local as well as distributed XML documents.

This transparency of distribution is a clear advantage to application programmers, as applications becomes easier and more simple to write. The same application that works in a local environment, works in a distributed environment.

Concurrency and transaction control

DOM Imperative programming updates data destructively. When data is shared between processes, concurrency control is necessary to ensure that data is not updated by processes, while other processes access or update the same data.

DOM implementations (including the used one) does not provide concurrency control of files being loaded from and saved to storage. Such control must be implemented by the application programmer.

Concurrency control is not implemented in the DOMDictionary, but should have been provided in methods `saveDict` and `loadDict`.

Transaction and concurrency control can be introduced by storing XML documents in databases instead of in flat files.

DVM `XMLStoreDictionary` does not contain any code for concurrency control. As modifications are made by creating new documents, accessed document data is never updated/changed. Documents can thus be accessed (read) by several processes without a need for concurrency control. Lines 73-74 show how modifications and saving are made, without any concurrency related code.

The name service functionality in XML Store introduces updateable variables. Updating name to value reference bindings in the name service introduces a possibility that updates may get lost. This is a shortcoming of the name service, which could be solved by implementing concurrency control.

A small scenario illustrates a situation in which a dictionary modification is lost. Two users A and B each insert a new word into the dictionary at the same time.

1. The dictionary document loaded in the `insert` method is the same for both XMLStoreDictionary applications.
2. The two different words are inserted dictionary document used within each `insert` method.
3. Both versions of the dictionary are saved.
4. The dictionary document name is updated with the value reference to the new dictionary document made by user A.
5. The dictionary document name is updated with the value reference to the new dictionary document made by user B.

The value reference to the dictionary document made by user A can not be retrieved anymore using the name service. The change made by A is "lost". We present a proposal for modified name service functionality, presented in section 7.2.4, that solves this problem.

Transaction control In the value-oriented programming model, documents are never lost, because modifying and saving documents does not delete old documents. Implementing simple transaction control becomes a simple task, because the process of *roll back* (retrieving the document version before changes were made) can be implemented by updating a variable (e.g. a cell or mutable node) containing a value reference for the modified document to contain the value reference for the old document.

7.2.4 Name service improvements

We showed how document updates can be lost. Such losses can be prevented by improving the name service functionality and implementing simple concurrency control within applications.

Concurrency control can be achieved by an approach called optimistic locking. Optimistic locking is based on the assumption that two processes seldom access the same value at the sametime. Transactions are therefore allowed to proceed as if no problems exist. If conflicts arise one or more transactions are aborted and must be restarted. The drawback of optimistic locking is the risk of starvation, where a process repeatedly has its transaction aborted and restarts.

Optimistic locking of values, in the value-oriented programming model, can be implemented as follows: A reference for the value *val* is kept in an updateable variable *upRef*. A new value *newval* is created. Updating the value and committing the change are made by setting *upRef* to to contain the value reference for *newval*, which is an atomic operation. Abortion of the update is made by setting *upRef* back to the reference for *val*.

The `bind` and `rebind` methods in the name service are modified to implement optimistic locking. In the central name service solution the optimistic locking occurs at the central server.

The functionality of `rebind` is modified to the following:

`void bind(String name, ValueReference ref)` Creates a binding between `name` and `ref`. The binding is shared with all other peers within the XML Store. If creation of the binding is not possible because, the binding already exist, an Exception is thrown

`void rebind(String name, ValueReference ref, ValueReference oldref)` This method updates a name-value reference binding, i.e. after having invoked `rebind(name, valref, oldvalref)`, `name` is no longer bound to `oldvalref`, by instead bound to `valref`. If updating is not possible (`oldvalref` is not equal to the actual reference bound to `name`), then a `ConcurrencyException` is thrown.

Using the modified name service can be illustrated for the `insert` method of `XMLStoreDictionary` as follows: The `insert` method is simply modified, such that the insertion is tried repeatedly until a rebind is successfully performed. The example has not been compiled and tested.

```
1    public void insert(Node word)throws DictionaryException{
2      ValueReference ref = xmlstore.lookup("dictionary");
3      if(!this.ref.equals(ref)){
4        dict = xmlstore.load(ref);
5      }
6
7      while(true){
8        Node kw = word.getChildNodes().getNode(0);
9        String keyword =
10         kw.getChildNodes().getNode(0).getNodeValue();
11
12       int index = binaryLookup(keyword);
13       if(index >= 0) {
14        throw new DictionaryException("Word exists!,
15                                       nothing inserted");
16       }
17
18       index = -index-1;
19       dict = DVMUtil.insertChild(dict, index, word);
20       ValueReference newRef = xmlstore.save(dict);
21       try{
22         xmlstore.rebind("dictionary", newRef, ref);
23         return;
24       }catch(ConcurrencyException e){
25         ref = xmlstore.lookup("dictionary");
26         dict = xmlstore.load(ref);
27       }
28     }
29   }
```

The improvement of the name service functionality can be used to implement simple concurrency control and thereby eliminate the possibility of loosing data.

7.2.5 Dictionary Extension

The dictionary application can be expanded to maintain a *search count* for each keyword in the dictionary document. The search count is the number of times a keyword has been requested. This feature may be used to provide lists of most popular and least popular words.

The search count is introduced by changing the structure of the dictionary document. Word elements are expanded to contain a third child element, an XML element with the tag name *count*. This element has no children but a single attribute named *value*. The attribute value contains the search count of the keyword contained within the word element.

The search functionality of dictionary applications is modified to update the search count each time a keyword has been searched and found. For the DOMDictionary and the XMLStoreDictionary presented in sections 7.2.1 and 7.2.2 this requires an update of their respective `keywordSearch` methods.

The DOMDictionary keywordSearch method is modified such that if a binary search is successful the search counter of the found keyword is incremented by one and the dictionary document is unparsed and saved to disk. This is illustrated below.

```
50   public Node keywordSearch( String keyword ){
51     loadDict();
52     Document doc = builder.newDocument();
53     if( binarySearch(keyword) ){
54       NodeList nlst =
55         ((Element)match).getElementsByTagName("count");
```

```
56        Element count = (Element)nlst.item(0);
57        int c = Integer.parseInt(count.getAttribute("value"));
58        count.setAttribute("value", String.valueOf(c));
59        saveDict();
60
61        Node n = doc.importNode( match, true );
62        doc.appendChild(n);
63        dict = null;
64        return doc;
65      }
66      Node res = doc.createElement("word");
67      res.appendChild(doc.createTextNode("No match on keyword " +
68                                         keyword));
69      doc.appendChild(res);
70      dict = null;
71      return doc;
72    }
```

The new implementation of the `keywordSearch` method in the DOMDictionary is straight forward. The imperative DOM interface makes it possible to change the attribute value of the count node by a simple `setAttribute` method call. The disadvantage of using the DOM interface is clearly that the whole document must be unparsed and saved to update the persisted dictionary document.

In the XMLStoreDictionary the nodes holding the search count are implemented using mutable nodes. The `keywordSearch` method is updated to modify the mutable nodes on successful searches.

```
39    public Node keywordSearch(String keyword){
40      ValueReference ref = xmlstore.lookup("dictionary");
41      if(!this.ref.equals(ref)){
42        dict = xmlstore.load(ref);
43      }
44      Node word;
45      int index = binaryLookup(keyword);
46      if(index >= 0){
47        word = dict.getChildNodes().getNode(index);
48        MutNode count = (MutNode)word.getChildNodes().getNode(2);
49        Node state = count.getNodeState();
50        int c = Integer.parseInt(state.getAttribute("value"));
51        Attribute attr =
52          factory.createAttribute("value", String.valueOf(c));
53        state = factory.createElementNode("count",
54                                          new Attribute[]{attr});
55        count.setNodeState(state);
56      }else{
57        word = DVMUtil.createElement("word",
58                                     "No match on keyword " +
59                                     keyword);
60      }
61      return word;
62    }
```

Updating the mutable "count" node requires creation of a new attribute, a new (immutable) node state and setting the node state. This happens in lines 51-55. The updating code therefore seems a bit more inconvenient, compared to the DOM version.

The XMLStoreDictionary application implementation on the other hand provides a considerable advantage. The dictionary document is not saved in order to modify the "count" node. The change of state in the mutable node is automatically saved, when `setNodeState` is invoked.

The `setNodeState` method is a destructive method of imperative nature.

Since XML Store does not provide transaction and concurrency control updates of the increments may get lost. The search count is not considered critical data, i.e. a few lost updates of a search count is tolerable. It can therefore be justified to use mutable nodes for the task.

7.3 Summary

We have implemented a dictionary application using the Document Object Model and another using the Document Value Model. The DOMDictionary application and the XMLStoreDictionary application, respectively.

The two different implementations have a similar document structure, as DOM and DVM provide similar structure of XML document and offer similar funcionality. DOM is, however a much large API than DVM.

However, the two applications differ. Evaluation of the DOMDictionary application showed how application programmers must handle memory issues, concurrency control, transaction control and any distributed aspects.

The XMLStoreDictionary application handles these issues. However the name server provides a flaw and introduces a potential concurrency problem. A simple correction of the name server functionality presented in 7.2.4 solves the problem.

The table below summarizes the evaluation of DVM (considering the improved name server) and DOM.

	DVM	DOM
Convenient interface which reflects treestructured data	+	+
Process arbitrarily large documents	+	-
Sharing of XML data	+	-
No parsing/unparsing before processing	+	-
Location transparency	+	-
Distribution transparency	+	-
Simple transaction control is implementable	+	-
Concurrency control not necessary when saving/loading	+	-

The table shows that the XML Store provides a range of advantages compared to the DOM interface.

We extended the dictionary applications with search counters. This also showed a considerable advantage of XML Store to the DOM interface. The mutable nodes of the XML Store makes it possible to introduce an imperative element into the value-oriented programming model. This makes it possible to update a single node without unparsing and saving the whole document as in DOM.

Generally the value-oriented programming model showed advantages over the imperative programming model when building distributed XML applications, as it allows applications programmers to remove attention from distribution logic. A program written for a non-distributed environment also works in a distributed environment. Furthermore, the value-oriented programming model should provide performance advantages as well, as value may be cached without concern to coherence, and whole documents do not have to reside in main memory before processing.

Chapter 8

Experimental results

Chapter 7 evaluated programming with the Document Value Model (DVM) compared to programming with the Document Object Model (DOM). It stated that DVM should have performance advantages. This chapter examines this by presenting a number of performance test and results.

We use the XML Store prototype implementation to examine performance, this is done through experimental measurements.

Section 8.1 tests initializing new XML Stores. Retrieval of document using XML Store and DOM is examined in section 8.2. Section 8.3 examines performance for accessing document content. Section 8.4 examines save performance and section 8.5 examines the performance when modifying a document.

8.1 Cold start

DVM applications, as the dictionary application presented in chapter 7, are expected to have better *cold start* times than DOM applications. Cold start time is time spent on initialization. DVM applications are expected to have better cold start times, as they do not need to process entire documents on initialization.

During the execution time of a Java virtual machine (JVM) XML Stores are only initialized fully once, i.e. the first time an XML Store is initialized. Any following initializations will share instances, such as the reference server, with the first initialized XML Store. These instances are therefore not initialized again. A cold start is therefore only performed for the first XML Store initialization within the same JVM.

Different aspects influence cold start of XML Store. These aspects are initialization of the reference server and initialization of the disk. Reference servers keep all value references to locator mappings in memory, these have to be loaded upon initialization. Initialization of disks are not expected to influence initialization time as nothing is loaded from disk on initialization.

Section 8.1.1 examines how disk size influence cold start time and section 8.1.2 examines the reference server size influence.

8.1.1 Disk initialization test

The disk is constructed such that the size of the disk file should have no influence on cold start time.

The disk initialization test examines the influence of the disk's size on cold start time by measuring the initialization time of XML Stores using disks with different sizes.

In order to examine the influence of the disk size on cold start time the influence of reference server initialization must be neglected. Reference server initialization is affected by the number of value reference to locator mappings in the reference server. Keeping the number of mappings in the reference server constant for all disk file tests, the reference server contributes with a constant factor to the time measurements.

The disk initialization test is carried out as follows:

1. 10 XML Stores with increasing disk sizes, but equal number of mappings in the reference server (nodes) files are created.

2. These XML Stores are cold started and time is measured.

Figure 8.1, shows the results. The test confirmed that disk sizes do not affect cold start, as cold start time is more or less constant for increasing disk sizes.

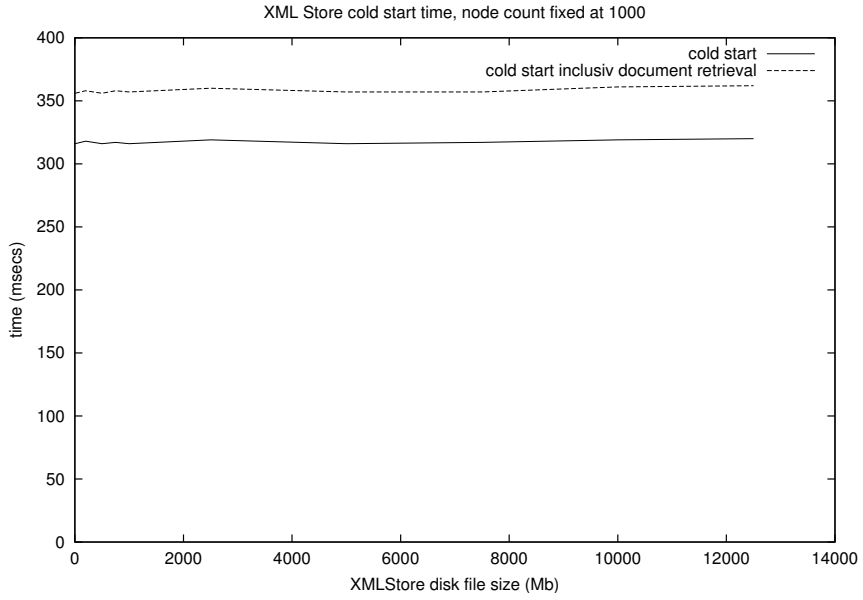


Figure 8.1: Cold start measures: XML Stores with a 1000 nodes, but increasing disk file sizes were produced. With the same amount of nodes, the XML Stores also contained the same amount of mappings. The cold start time was measured as a function of disk file size. The time measures were produced from an average of 10 test runs.

8.1.2 Reference server initialization test

As mentioned the number of value reference to locator mappings persisted in the reference server are expected to influence cold start time. We expect that there is a linear relationship between the two, as each mapping has to be inserted into the reference server.

The reference server initialization test examines how the number of persisted value reference to locator mappings affects cold start time. It is done by measuring the initialization time of XML Store using reference server files of various sizes.

The test was carried out as follows:

1. 10 XML Stores, all with the same disk file size, but increasing number of nodes (increasing number of value reference to locator mappings) were created.

2. The cold start time was measured for each of the 10 XML Stores using the created disk and reference server files.

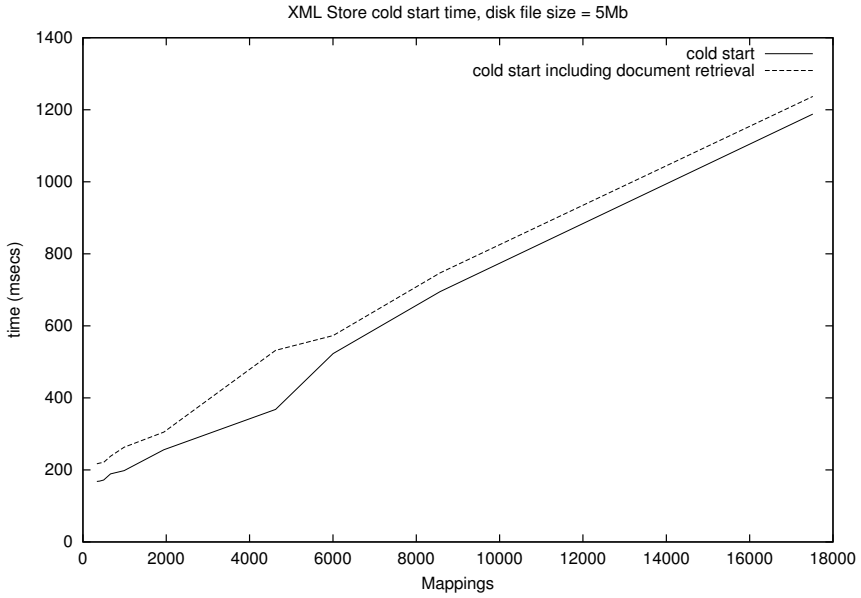


Figure 8.2: Cold start measures: XML Stores with a disk file size of 5 Mb, but different number of mappings were produced. The cold start time was measured as a function of the number of mappings. The time measures were produced from an average of 10 test runs.

The results are shown in figure 8.2. The test showed that the number of value reference to locator mappings affects cold start time. When the number of mappings increases, the cold start performance time increases. The relation between the number of mappings and the cold start time appears to be linear.

During the reference server initialization a file holding existing mappings is parsed and each mapping is inserted into the reference server. The more

mappings in the file, i.e. the bigger the file, the longer parse time. This is confirmed by the test result.

8.1.3 Summary

The influence of disk size and of reference server size on cold start time has been examined. The test results clearly show that the disk size has no influence on cold start time, whereas reference server size has influence on cold start time. The cold start time increases with increasing number of value reference to locator mappings in the reference server.

8.2 Document retrieval

The strategy for document retrieval is fundamentally different in the implemented XML Store compared to DOM implementations. This should provide XML Store with a performance advantage compared to DOM when retrieving documents.

Document retrieval time is the time it takes before you can start working on a document. Retrieving an XML document, using XML Store, means loading the root node from disk to main memory. Document retrieval using DOM, means parsing a serialized version of the XML document from some external source into main memory.

The differences in performance of document retrieval in XML Store and DOM implementation are illustrated by examining performance of document retrieval in both. Section 8.2.1 examines XML Store and section 8.2.2 examines DOM.

8.2.1 XML Store

Loading an XML document from XML Store is done lazily, i.e. XML elements and character data are loaded when needed. This affects the time spent on document retrieval.

The document retrieval test is done by measuring the time it takes to 1) use a name and the name service specified by the Document Value Model (DVM) (see chapter 4) to lookup a value reference and 2) use the value reference to retrieve the document root node.

With these properties we expect that neither document size nor the number of nodes in documents (number of mappings in the references server) affects time spent on document retrieval.

The test was carried out by testing document retrieval on different documents. The test documents from the cold start test (see section 8.1) was reused.

The results are shown in the figures 8.1 and 8.2. The test results showed that document retrieval was not dependent on document size nor the number of nodes in the document.

That document retrieval time is not influenced by document size nor number of nodes is simple to explain. The implementation of XML Store is made such that the content of root nodes are not loaded on document retrieval. Instead a proxy node is returned for the root node (see section 6.1.1). Since returning

a proxy for the root node performs equally for all root nodes, the document retrieval time is unaffected by document content.

8.2.2 Document Object Model implementations

Startup properties of the XML Store implementation differ from the startup properties of DOM implementations. As described, using DOM, a serialized XML Document must be parsed into an in memory representation, before any DOM application can "start". Therefore it make no sense to differentiate between DOM upstart time and DOM document retrieval time.

The different properties of XML Store and DOM make it difficult to compare upstart performance. However, testing the upstart performance of a DOM implementation will illustrate the different properties.

The upstart of the Document Object Model implementation is the time it takes to parse an XML document. [1]

Upstart type of two groups of XML documents is measured. The first group of XML documents has a fixed number of nodes, while their *size* vary. The size of an XML document is the disk space it uses when serialized. The second group of XML documents has a fixed size, while the number of nodes varies. Each test is made by producing retrieval times for 10 XML documents, with increasing size and increasing number of nodes, respectively.

As whole documents are parsed into memory we expect that upstart time is affected by document size. The number of nodes in a document should also affect upstart time as each node must be parsed into an internal representation.

The results are shown in figures 8.3 and 8.4. The test result for the group of XML documents with fixed node count shows that upstart time of the used DOM implementation increases as the document size increases. The relation between document size and upstart time appears to be linear.

The test result for the group of XML documents with fixed size shows that upstart time of the used DOM implementation increases as the node count increases. The upstart time appears to be linear for increasing node counts.

The results also show that for small node counts upstart time is decreasing. The documents with a small amount of nodes is created by producing CDATA nodes with large amount of data. A possible explanation for the decreasing upstart time is that the DOM implementation is slow when producing such nodes, i.e. nodes with large CDATA sections.

The tests show that upstart times of the used DOM implementation depends on both the document size and the number of nodes in the document.

8.2.3 Summary

The document retrieval time using XML Store is not affected away by the document size of number of nodes in the document. Compared to the performance of document retrieval using a DOM implementation, XML Store has an advantage. Upstart time using DOM time is affected by both increasing document size and increasing number of nodes.

[1] The test is not an actual test of DOM as this is an interface specification. It is a test of the parser used for building (parsing) the serialized XML document. The Xerces parser [37] was used for the test.

XML Store properties are well suited for applications similar to the dictionary application (presented in chapter 7), where only small parts of available data is used. With other types of applications that always process entire documents, the DOM approach is better. Extending the XML Store interface (and implementation) with an eager-load function (that is a load function that always loads entire documents) would solve this issue.

8.3 Loading Document

As documents are loaded lazily, it is interesting to look at performance of access to documents. Furthermore it is interesting because nodes should only be loaded from disk once, and then put into a cache, i.e. accessing the same node multiple times should be fast.

Document access means loading node type, value, attributes and children from disk to main memory. The document access performance is evaluated by measuring the time it takes to access document data.

XML documents can reside on any peer in the XML Store. They can reside locally or on several other peers. The access performance depends on where XML documents are stored.

The document access performance is evaluated for an XML dictionary document residing on a local peer and an XML dictionary document residing on several peers.

Section 8.3.1 examines document access on a local disk. Section 8.3.2 examines document access for an XML dictionary residing on several peers.

8.3.1 Local access

Document access performance depends on the load performance, i.e. the time it takes to load document parts. Loading documents residing locally is expected to be faster than loading document from other XML Store peers.

The access performance of documents stored locally is evaluated by measuring the time it takes to access (load from disk to main memory) data in a document stored locally.

The dictionary application presented in section 7.2 is used for test. The test is made by storing the dictionary XML document on the actual peer and measure the time needed for performing a keyword search in the dictionary. The keyword search is performed 10 times for the same keyword (the keyword is 'foo' and exists in the dictionary).

The results can be seen from table 8.1. The test result clearly shows that the search time (or access time) is higher for the first search, than for following searches.

That the search time is much longer for the first keyword search shows that the caching scheme works. During the keyword search in the dictionary lots of nodes are accessed. These nodes are all cached (in main memory) when being accessed for the first time. Subsequent searches simply uses the cached version.

Access time may be improved by loading more than one node at a time (load buffering).

Search #	Time
1	3780
2	2
3	5
4	2
5	2
6	2
7	2
8	2
9	3
10	4

Table 8.1: A keyword search was performed 10 times in a row using the XMLStoreDictionary application. The dictionary is FOLDOC and the keyword was 'foo'. The dictionary file resided at the XML Store peer on which the keyword search were performed. The FOLDOC dictionary i 7.6 Mb

8.3.2 Several peers

As mentioned in section 8.3.1 document access depends on where documents are stored. Loading documents stored on other XML Store peers is expected to require more time, than loading from local storage.

As in section 8.3.1 the dictionary application is used for the evaluation. The dictionary XML document is stored on 3 XML Store peers, i.e. on 3 different computers. Keyword search is performed from a fourth XML Store peer having no dictionary document. This peer will therefore be forced to load the document from the 3 other peers. All XML Store peers resides on the same intranet.

The test result is shown in table 8.2. The test result again shows, that the first search takes more time than the following nine searches. The test also shows that the search time for the first search is higher, when loading the document from other peers, compared to when loading the document on from a local storage.

Just as when performing a search on a dictionary document stored locally, the search time is highest for the first search. This time the first search requires more time, because the dictionary document is loaded from other XML Store peers.

Using techniques such as buffered loading (see section 5.2.4), chould help speed up load access.

8.3.3 Summary

The load time differs when loading documents from the local peer and loading distributed from other peers. When loading from the actual peer, the performance is low but acceptable. The load performance might be improved by in-lining and load buffering.

Search #	Time
1	57807
2	4
3	4
4	4
5	5
6	6
7	4
8	6
9	4
10	4

Table 8.2: Keyword using the XMLStoreDictionary application. The FOLDOC dictionary resided on each of 3 XML Store peers. The keyword search was performed from a fourth peer not containing any parts of the FOLDOC dictionary.

8.4 Saving Documents

Saving XML documents is central to working with XML Store.

The save performance of the XML Store prototype implementation is examined by measuring the time needed to save documents.

In the prototype implementation, document save times are expected to depend on the size of the document being saved. The save times (or save performance) are on the other hand expected to be unaffected by the data amount, already saved on disk.

Section 8.4.1 examines save times of different document sizes. Section 8.4.2 examines save times on disks with different sizes.

8.4.1 Save functionality

The prototype implementation of XML Store provides simple save functionality without any buffering.

In the prototype implementations values are written to disk, when the save method is invoked on the values. The performance of this save functionality is tested by measuring the save time for XML documents of different sizes.

The test is performed by creating 10 XML documents of different sizes. Each XML document is saved in an empty XML Store, i.e. an XML Store containing no documents.

The test results shows that save times increase with increasing XML document size. The results are illustrated by figure 8.5.

8.4.2 Disk access

Values are saved log-structured on disk and random access to the disk is therefore not necessary for saving values. The disk access time is therefore to be constant. The save time should therefore only be influenced by the amount of data saved.

The disk save performance is tested by measuring the save time of the same XML document in XML Stores with different disk sizes.

The test is performed by creating 10 XML Stores with different disk sizes. The same XML document is saved in each of the XML Stores and the save time is measured.

The results are shown in figure 8.6. The test results show, that the save time increases with increasing disk file size.

This result contradicts the theory of log-structured filesystem, in which save times should only be affected by the amount of saved data.

The results also show that the increase in save time from saving in an empty XML Store to saving in an XML Store with disk size of 12.5 Mb is only approximately 350 Msecs.

The reason for the test result is not clear, but a possible explanation may be found in the use of Java's `RandomAccessFile` for writing data to disk. Although log-structured save behavior can be simulated using this class, the save performance achieved by using the class, may be affected by the amount of already saved data.

8.4.3 Summary

The XML Store save performance is affected by the size of the saved document, such that save times increase with increasing document size.

The save performance is however also affected by the amount of data already saved on disk.

Introducing asynchronous save functionality, as described in section 5.2.4, will shorten the delay of saving documents, as documents can be saved as a background process.

8.5 Document modification

The value oriented programming model provides XML Store with an advantage when modifying documents compared to modifying documents in an imperative programming model.

Document modification in the XML Store is 1) changing the document 2) saving the resulting new document. The performance of document modification is tested by measuring the document modification time.

In the performance test the save time of the XML document being modified is first measured. Thereafter a small XML subtree is inserted into the document and the modification times are measured. These can be seen from table 8.3.

Document Size	Save Time	M Insert Time	M Save Time	Sum
5 DB	132 Msec	19 Msec	3 Msec	22 Msec

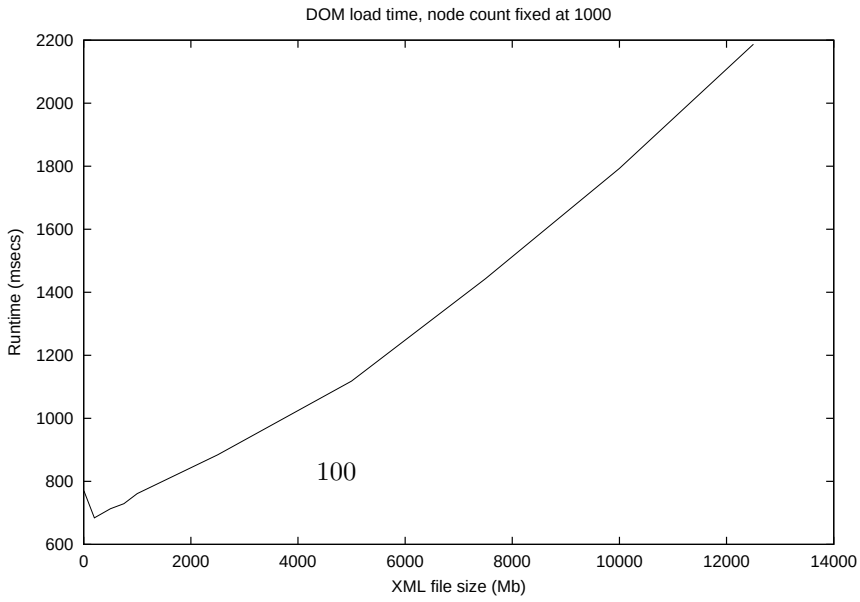
Table 8.3: Modification of an XML document. The second column contains the time for saving the whole XML document. The third to the fifth columns contains the modification times, first the time to insert new XML data, secondly the time to save the modified document and third the sum of the two.

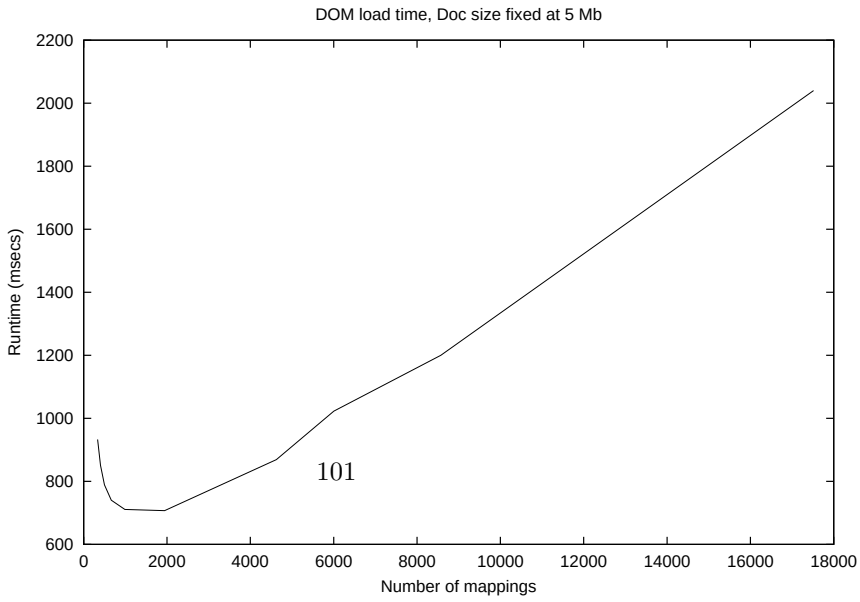
The test results show how document modification in XML Store is faster than saving the whole document. This is a result of the share-create style (see

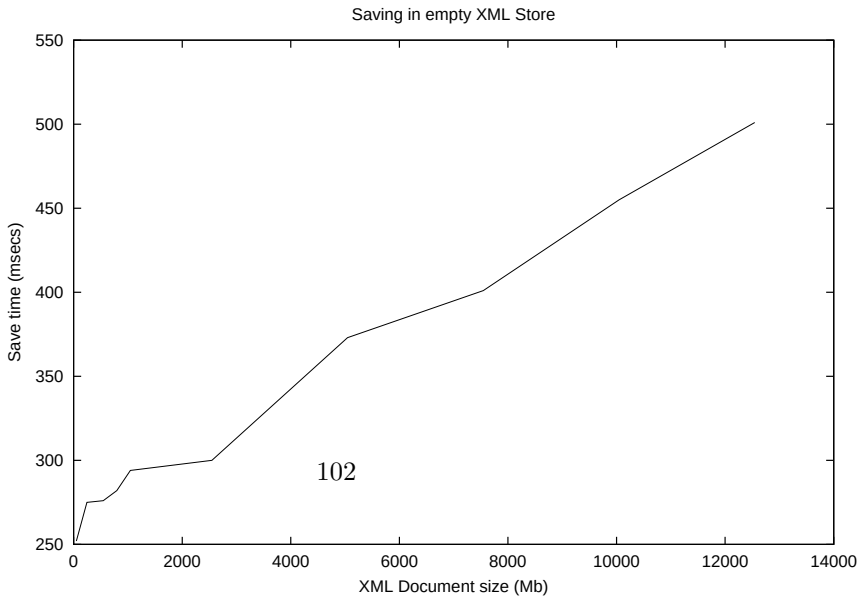
section 3.2). The non modified parts of the document are shared and not saved again.

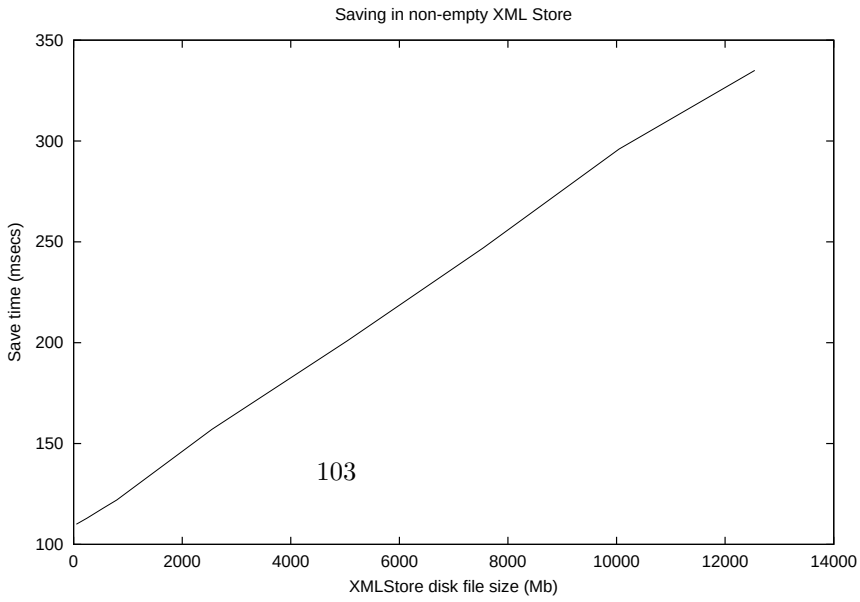
The test results also show, that the most timeconsuming operation in the modification is inserting the new XML subtree.

As described in section 6.1.4, the XML prototype implementation represents child nodes as arrays. Insertion and removal of child nodes in such a representation becomes expensive when the number of child nodes is high. The document used in the test has a root with 335 child nodes, and the inserted subtree is added as a child to the root node. As described in section 6.1.4 a different strategy for representing child nodes could improve modification performance.


DOM load time, node count fixed at 1000
Runtime (msecs)
XML file size (Mb)
2200
2000
1800
1600
1400
1200
1000
800
600
0
2000
4000
6000
8000
10000
12000
14000
100


DOM load time, Doc size fixed at 5 Mb
Runtime (msecs)
Number of mappings
2200
2000
1800
1600
1400
1200
1000
800
600
0
2000
4000
6000
8000
10000
12000
14000
16000
18000


Saving in empty XML Store
Save time (msecs)
XML Document size (Mb)
550
500
450
400
350
300
250
0
2000
4000
6000
8000
10000
12000
14000


Saving in non-empty XML Store
Save time (msecs)
XMLStore disk file size (Mb)
350
300
250
200
150
100
0
2000
4000
6000
8000
10000
12000
14000

Chapter 9

Future works

Some aspects of XML Store were beyond the scope of this thesis. These aspects are however important for future work of the XML Store. The work presented in this thesis may be improved and extended in various aspects:

9.1 Name Service

The name service presented is inadequate in a number of ways. Improvements could be made in three areas:

The concurrency problem presented in chapter 7 should be solved for the name service to be usable. A solution to the problem has been proposed, see section 7.2.4.

Effort could be made towards removing the centralized solution. The name service should be implemented as a decentral distributed service to eliminate the single point of failure problem. More effort on how this may be done is needed.

Finally effort could be made towards removing the limited name space offered by the name service. More advanced management of names is needed.

9.2 Distributed Garbage collection

The current implementation of the XML Store disk layer only allows saving and loading documents. Deleting documents from disk storage is not possible. As a consequence storage will gradually be filled up, possible with data no longer used. A distributed garbage collector will solve this problem.

Distributed garbage collection can be defined as:

The aim of a distributed garbage collector is to ensure that if a local or remote reference to an object is still held anywhere in a set of distributed objects, then the object itself will continue to exist, but as soon as no object any longer holds a reference to it, the object will be collected and the memory it uses recovered[5]

In a value oriented context 'object' can be exchanged with 'value'. Implementing distributed garbage collection is a complicated task, but it has been done. One way to implement distributed garbage collection is through leases. Using leases is a simple approach keeping data alive, if a leases have not been renewed in a specified period of time the data may be removed. It is up to the

user of the data to renew leases. Another approach is reference counting where some process keeps track of how many references exist to each object/value, when the number is zero the object/value may be removed.

9.3 Network communication

The process of locating documents in XML Store relies on IP Multicast, which is not reliable or scalable. Messages sent on a network may therefore get lost. IP Multicast should therefore be exchanged with a reliable multicast [**?**], or another routing algorithm e.g. Chord as presented in Baumann et al [9].

Loading distributed data eagerly (as opposed to the present lazy approach) may in some situations improve performance. Loading values eagerly is however a somewhat complicated affair. Documents may be distributed among several peers, this make it difficult to implement a simple "read-ahead" strategy, as it is difficult to know where to read from. More research need to be put into this aspect.

Furthermore effort could be made in making the XML Store load and save protocol asynchronous.

9.4 Mobility

The world is being increasingly populated by small and portable computing devices, including laptops, handheld devices such as personal digital assistants (PDAs), mobile phones, etc.

If XML Store should support such computing devices, it must be able to support mobility of data.

XML Store and the Document Value Model interface provides an interface where document location is transparent. The Document Value Model therefore supports mobility of data. This has, however, to be implemented in the XML Store.

9.5 Querying XML documents

As increasing amounts of information are stored, exchanged, and presented using XML, the ability to intelligently query XML data sources becomes increasingly important.... An XML query language must provide features for retrieving and interpreting information from diverse sources of XML documents[16].

Implementing a query language for XML Store would provide considerable advantages to application programmers, as it eases the process of extracting information from XML documents.

XQuery and *XPath* [39] are W3C standards for expressing database style queries of XML documents. With XPath, a set of nodes can be selected based on a regular path expression. XQuery is a query language using XPath to express queries. Effort could be placed in implementing XQuery for querying documents in XML Store.

Bibliography

[1] W3C team. Extensible markup language (xml) 1.0 (second edition). http://www.w3.org/TR/REC-xml.

[2] The xml information set. http://www.w3.org/TR/xml-infoset/.

[3] W3C team. Document object model (dom) level 2 core specification. http://www.w3.org/TR/2000/REC-DOM-Level-2-Core-20001113/.

[4] Sax: Simple api for xml. http://www.saxproject.org/.

[5] George Coulouris, Jean Dollimore, and Tim Kindberg. *Distributed Systems Concepts and Design*. Addison-Wesley, 2001.

[6] Napster. http://www.napster.com/.

[7] The Gnutella protocol specification v0.4. http://www.gnutella.co.uk/-library/pdf/gnutella_protocol_0.4.pdf.

[8] I. Clarke, O. Sandberg, B. Wiley, and T. W. Hong. Freenet: A distributed anonymous information storage and retrieval system. In *Proceedings of the Workshop on Design Issues in Anonymity and Unobservability*, pages 46–66, 2000.

[9] Mikkel Fennestad Tine Thorn, Anders Baumann. A distributed value-oriented xml store. Master's thesis, IT University of Copenhagen, July 2002.

[10] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns Elements of Reusable Object-Oriented Software*. Addison Wesley, 1999.

[11] Michael I. Schwartzbach Anders Møller. The xml revolution, technologies for the future web. Anders Møller, Michael I. Schwartzbach.

[12] What does each dom level bring? http://www.mozilla.org/docs/dom/reference/levels.html.

[13] http://www.w3.org/TR/1998/WD-DOM-19980720/cover.html.

[14] Jason Hunter and Brett McLaughlin. Jdom. http://www.jdom.org.

[15] Chapter 17 of the *xml bible, second edition : xsl transformations*. http://www.ibiblio.org/xml/books/bible2/chap17.html#d1e495.

[16] Xquery 1.0: An xml query language. http://www.w3.org/TR/xquery/.

[17] Ronald Bourret. Xml and databases. http://www.rpbourret.com/xml/XMLAndDatabases.htm, 2002.

[18] P. Druschel and A. Rowstron.

[19] Frank Dabek. A cooperative file system. http://citeseer.nj.nec.com/dabek01cooperative.html.

[20] W3C. Overview of sgml resources. http://www.w3.org/MarkUp/SGML/.

[21] Xsl: Extensible stylesheet language. http://www.w3.org/Style/XSL/.

[22] Simple object access protocol (soap). http://www.w3.org/TR/SOAP/.

[23] The extensible hypertext markup language (xhtml). http://www.w3.org/TR/xhtml1/.

[24] Scalable vector graphics (svg). http://www.w3.org/TR/SVG/.

[25] Minimal xml. http://www.docuverse.com/smldev/minxml.jsp.

[26] Foldoc: Free on-line dictionary of computing. http://www.foldoc.org.

[27] D. Suciu S. Abiteboul, P. Buneman. *Data on the Web*. Morgan Kaufmann, 2000.

[28] Javatm 2 platform, standard edition (j2setm). http://java.sun.com/j2se/1.4/index.html.

[29] Xalan-java version 2.4.d1. http://xml.apache.org/xalan-j/index.html.

[30] Morten Primdahl. Querying the web http://www.it-c.dk/∼morten/thesis/. Master's thesis, IT-University of Copenhagen, 2002.

[31] Ronald Bourret. Mapping dtds to databases. 2001.

[32] Fritz Henglein. Desiderata for an applicative persistent store manager (xml store). unpublished, memo, 2001.

[33] William Stallings. *Cryptography and Network Security: Principles and Practice*. Prentice Hall, second edition, 1998.

[34] W3C team. Hypertext markup language (html) home page. http://www.w3.org/Markup.

[35] Mendel Rosenblum and John K. Ousterhout. The design and implementation of a log-structured file system. *ACM Transactions on Computer Systems*, 10(1):26–52, 1992.

[36] Dictionary.com. http://www.dictionary.com.

[37] Xerces2 java parser readme. http://xml.apache.org/xerces2-j/index.html.

[38] Fabrice Le Fessant.

[39] Xml path language (xpath) version 1.0. http://www.w3.org/TR/xpath/.

[40] Peter Sestoft. Grammers and parsing with java. http://www.dina.dk/∼sestoft/programmering/parsernotes.pdf, 01 1999.

[41] David Mertz. Lightweight xml libraries. http://www-106.ibm.com/developerworks/library/x-tiplwt.html/.

[42] David Mertz. Xml matters: Transcending the limits of dom, sax, and xslt. http://www-106.ibm.com/developerworks/xml/library/x-matters14.html.

[43] Bijan Parsia. Functional programming and xml. http://www.xml.com/lpt/a/2001/02/14/functional.html.

[44] Feng Tian, David J. De Witt, Janjun Chen, and Chun Zhang. The design and performance evaluation of alternative xml storage strategies. *SIGMOD Record*, 31(1):5–10, March 2002.

[45] Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest. *Introduction to Algorithms*. The MIT Press, 1998.

[46] J. Hughes. Why Functional Programming Matters. *Computer Journal*, 32(2):98–107, 1989.

[47] Jim Waldo, Geoff Wyant, Ann Wollrath, and Sam Kendall. A note on distributed computing. In *Mobile Object Systems: Towards the Programmable Internet*, pages 49–64. Springer-Verlag: Heidelberg, Germany, 1997.

[48] Dennis M. Ritchie. The develpment of the c language. http://cm.bell-labs.com/cm/cs/who/dmr/chist.html, 1996.

[49] Xml: Proposed applications and industry initiatives. http://www.oasis-open.org/cover/xml.html#applications.

[50] Dimitre Novatchev (dnovatchev@yahoo.com). The functional programming language xslt - a proof through examples. 2001.

[51] Fritz Henglein. Xmlstore, specification and reference implementation of core xml api. unpublished, memo, 2001.

Appendix A

Property file

Using XML Store requires initializations specific to disk functionality and network functionality to be performed.

XML Store *properties* are values used for initializing an XML Store. The properties are stated in a *property file*, using Java default property handling API. Upon initialization of XML Store properties are read from the property file.

The properties files follow the syntax required when using the Java API to access property files.

- Lines starting with '`#`' or '`!`' indicate comments and are therefore ignored.
- Every other non empty line contains a property. Property names contain no spaces and are the first word within a line. They are separated from their property value by '`=`', '`:`' or whitespaces. The property value is the remaning line.

A more detailed syntax description of Java property files can be found in the Java API [28]. This description surfice for the XML Store property file.

The XML Store properties are:

`Factory.` The factory class implementing the `XMLStoreFactory` interface. By chosing this class the XML Store implementation is chosen.

`DiskFactory` The factory class needed to create disks.

`Disk.` The disk composition. Disks are separated by '`;`'. The disk first disk is considered the "outer disk". Currently only `GlobalDisk` and `LocalDisk` exist. Stating `GlobalDisk` is followed by a port number, which is used for network communication.

`Name_server_IP.` IP address of the central name server.

`NameServerPort.` Port number which the central name server listens on.

`Server_IP.` The IP Multicast address used by the reference server. This identifies the multicast group.

`ReferenceServer_port.` The port number, which the reference server peer listens on.

`Network_timeout`. Time out set when sending lookup requests using the name server or reference server. If answers are not received within the value of `Network_timeout`, it is taken as the looked up key does not exist. The time out is stated in milliseconds.

The following is an example of a property file:

```
# Factory class used to create XMLStore objects and Node objects
Factory = edu.it.dvm.DVMXMLStoreFactory

#DISK INFORMATIONS:
#disk factory
DiskFactory = edu.it.disk.DiskFactory

#disk
Disk = GlobalDisk 1111; LocalDisk
# AsynchronousSaveDisk also available

# SERVER INFORMATIONS:
#name server ip
Name_server_IP = 127.0.0.1

# Port which the server is listening at:
NameServer_port = 2222

# IP of the server
Server_IP = 228.5.6.7

# Port which the ReferenceServer is listening at:
ReferenceServer_port = 2224

# Network timeout
Network_timeout = 2000
```

Appendix B

Samples

B.1 FOLDOC Dictionary

The dictionary used for the evaluation of DOM, SAX and XSLT is the Free On-line Dictionary of Computing (FOLDOC) dictionary [26], which contain definitions for computing terms.

The dictionary source is available in a plain text file. This file has the following properties.

- The words are sorted alphabetically.
- The words are unique, that is a word is not defined twice.

The file has been converted into serialized a XML representation still conforming to the above properties. The XML structure is defined by a DTD presented section B.1.1.

B.1.1 Dictionary DTD

```
<!ELEMENT dictionary (word+)>
<!ELEMENT word (keyword, desc?)>
<!ELEMENT keyword (#PCDATA)>
<!ELEMENT desc ( p+ )>
<!ENTITY %text "#PCDATA | type | link | ref">
<!ELEMENT p ( %text; )* >
<!ELEMENT type (%text;)*>
<!ELEMENT link (%text;)*>
<!ELEMENT ref (%text;)*>
```

Each word (computing term) have a `word` element containing both the word and its description. The word is put in a `keyword` (keyword) element and the description in a `desc` element. The root element, called `dictionary`, contains `word` elements. Example 2.1 presents a sample of the serialized representation, only the word “foo” is shown.

B.2 Source code

Java applications using the serialized XML representation of the FOLDOC dictionary have been build. These are `SAXDictionary`, `DOMDictionary` and `XSLDictionary`. The serialized XML representation of FOLDOC have also been saved in XML Store and an application, `XMLStoreDictionary`, using this XML document have been developed. `NodeCounter` is a simple application counting nodes in XML documents, which are store in XML Store. The following pages contain source code for these sample applications.

Appendix C

Source code

The source code for the XML Store prototype implementation is listed the following pages.