Master Thesis

Value-oriented XML Store

Jesper Tejlgaard Pedersen, c960709
The Technical University of Denmark

Kasper Bøgebjerg Pedersen
IT-University of Denmark

Supervisors:
Michael R. Hansen
Fritz Henglein

August 2002

Abstract

XML Store is a distributed value-oriented storage facility for storing XML documents. XML documents stored in XML Store can be accessed and manipulated using the Document Value Model (DVM). This thesis illustrates that such a storage facility can be constructed. The value-oriented programming model, proposed have shown advantages over traditional imperative models for working with persisted and distributed XML document. These advantages are easy caching and replication (as no coherence protocols are needed), lazy loading of documents, transparent persistence and distribution and sharing of documents.

Acknowledgements

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

The guidance and advices (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 provided the initial inspiration for the thesis, which we are grateful for.

During the period of the thesis much time have been spent with the newly graduated candidates 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.

A special thanks goes to Brian Christensen for spending his vacation on helping with setting up the report and to Thejs W. Jansen, Martin Skøtt, Petar Kadijevic and Kinnie Bak Pedersen for reviewing of the different versions of the thesis and giving valuable feedback.

Preface

This Master Thesis concerns the development and implementation of a value-oriented Application Programming Interface (API), which is convenient for access, manipulation and storage of 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.

Michael R. Hansen from the Institute of Informatics and Mathematical Modeling (IMM) at the Technical University of Denmark and Fritz Henglein from the IT-University of Denmark 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 Jesper Tejlgaard Pedersen.

The thesis has been started on the 1st of February 2002. It has been submitted by Jesper Tejlgaard Pedersen on the 31st of August. Kasper Bøgebjerg Pedersen will continue working on and improving the thesis until his submission on the 1st of November.

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) is written in the programming language Java. Being able to read Java code is thus essential for the reading of the thesis. Besides Java basic knowledge of functional programming languages such as ML may provide a small advantage.

The appendixes referred to through out the thesis can be found separately. The thesis and source code is available on the web site:

`http:www.it-c.dk/people/tejl/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, SAX provides an event driven model for accessing XML documents. SAX is basically a lexer for XML documents. Neither SAX nor DOM (level 1 and 2) specifies an API for accessing persistent XML documents. Current DOM 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.

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* offer 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. Examples of client/server systems are web servers that handle requests for 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 that client/server systems as they eliminate the "single point of failure". Further, 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.

Building distributed systems is considerably more complex than building non-distributed applications. Application programmers need to handle distributed concerns as well as application logic. Applications programmers are required to deal with issues as when and how to cache data and when to use remote references.

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

[1]This does not apply to Napster.

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. Consistency of data is a general concern in systems where data are shared, complex transaction mechanisms are introduced to help keep data valid.

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. Such a programming model is in contrast to the imperative programming model.

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. Updations are performed by updating variable, which holds a reference to a value, with a reference to another value. This can be done automically.

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.
- Automatic 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 atomicly.

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 fact that XML documents can be traversed 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. Building basic operations and extending these is used as a general design principle.

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 list 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 (e.g. operations similar to those offered by DOM[3]). The API must be value-oriented.

Hide location and distribution Application programmers using XML Store should be oblivious to the location of XML documents. That is documents should, from 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

The desiderata of mentioned in the previous section does not include all necessary aspect of building XML Store. Additional aspect not necessary, but feasible to implement using XML Store exist.

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 a 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 includes the name server and the representation of child nodes. The problems with the implementations are analyzed and better solutions are proposed.

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 of the Extensible Markup Language (XML), presents different approaches for working with XML and states shortcoming 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 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) each 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.

Other APIs such as JDOM[14] offer Java programmers with a more convenient interface, which resembles DOM conceptually.

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. This is done in Xalan-Java, which is an application making XSLTs .

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

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 requires 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.

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 Distributed, value-oriented XML Store[9] provides location transparent storage of data, with efficient location independent *routing* in a decentralized environment. Routing is sending messages in a distributed network, from one network location to another, until the destination location is reached.

1.5 Conclusion

This thesis presents value-oriented interface, the Document Value Model, and a storage manager, XML Store, for distributed persistens of XML document.

The value-oriented Document Value Model (DVM) interface allows applications to create, access, modify and persist XML document. The DVM interface resembles the object structure of DOM, except that all aspect of the interface is value-oriented - any document modifications results in new documents, leaving the former document unchanged, except for mutable nodes.

DVM is not as vast and complex interface Document Object Model (DOM). Instead only the most basic (and needed) functionality is provided. More convenient functionality can be implemented by extending the interface or by introducing utility functions. Within this thesis utility functionality have been implemented.

Programming with DVM have considerable advantages compared to programming with the imperative DOM interface. In contrast to DOM nodes can be shared within and between documents in DVM, and persistence is location and distribution transparent and therefore not necessary to consider when building applications. The need for transaction control are removed using DVM, since updates are performed by a simple atomic set operation.

XML Store is build as a peer-2-peer network, in which single peers contain no information of other peers. XML Store is thus highly flexible to new peers joining the network.

Distributed lookup of documents is based on the IP Multicast communication. Using IP Multicast allows peers to contain no information on other peers. It is however not reliable, a reliable multicast service can be build.

XML Store have the Document Value Model interface and therefore gain the above described advantages of the interface.

The XML Store strategy for loading document enables handling of arbitrarily large document. Memory usage is low as document are loaded lazily (on request) and discarded again (from memory), when not used. This is transparent to applications programmers, such that document location makes no difference in applications, i.e. if documents reside in memory, on disk or across a network is transparent.

XML documents are not only shared, when loaded into memory, all nodes on disk or located on other peers may be transparently shared.

XML documents can be named using a name service functionality. This service is a implemented as a central server solution, which lacks of a limitted name space and no possibility to implement transaction control. More work are required to provide a suitable name server.

XML Store and its Document Value Model have been evaluated through implementation of simple applications, one counting nodes in an XML document and the other implementing a dictionary application. These illustrated the above mentioned advantages of value-oriented programming. Applications better suited for illustrating the advantages with regards to sharing of documents could, however, have been choosen.

The XML Store performance have been evaluated through a number experimental tests. These tests illustrated advantages of the value-oriented programming model. Modifying persisted documents and storing resulting new documents are fast compared to simply storing the documents. This is an effect of sharing. The simple caching strategy applied improves the performance of XML Store greatly. The tests also illustrated high loading times for distributed documents, which requires further work to improve the distributed load performance.

We have shown value-oriented application programming interface eases development of distributed 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 transaction control by provide atomic updates and provides transparent location and 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 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 in between the tags). 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 *sub elements*,

```
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]

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, that is the same attributes can only occur once in the same element, attributes can not be nested and they may only hold character data.

Mixed contents allows 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 just described, 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, discusses different approaches used to store XML documents.

2.1 XML documents as trees

As mentioned 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 to be well-formed when

1. It only contain 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.

The requirement of well-formedness basically just ensures that serialized XML documents can be transformed into labeled trees [27, p.29].

Figure 2.2 illustrate 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 a model of labeled ordered trees (directed acyclic graphs (dags) when sharing of subtrees is considered), when nothing else is noted.

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

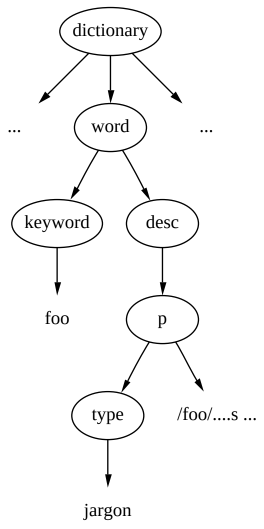


Figure 2.2: Labeled tree representation of XML document. Root element `dictionary` has child elements `keyword` and `desc` (... symbolizes more nodes)

writing programs that consider XML documents as a data source, i.e. programs 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 the two distinctive 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, e.g. XML documents, and is the official proposal from W3C. 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 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 Application programming interface (API), 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*. The DOM tree an abstract representation of the XML document it models. Although not part of the specification, DOM implementations[1] require that the entire XML document is parsed into memory. Once the document is in memory, the DOM interface provides tree based access to nodes in the DOM tree and methods for document manipulation.

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

To illustrate and evaluate DOM a sample application is implemented. This application is, as mentioned, a dictionary that supports (keyword) lookup and inserts of new words (see B for description and full source code). The XML document used 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 is spent building a result, on successful search the node representing the found word must be cloned (line 12, `importNode` clones), as DOM does not allow sharing of nodes. The node must be 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

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

```java
public Document keywordSearch( String keyword ){
  DocumentBuilderFactory factory =
      DocumentBuilderFactory.newInstance();
  factory.setIgnoringElementContentWhitespace(true);
  builder = factory.newDocumentBuilder();
  Document dict = builder.parse( new File( dictName ) ) ;

  Node match = binarylookup( dict, keyword );

  Document doc = builder.newDocument();
  if( match != null ){
    Node n = doc.importNode( match, true );
    doc.appendChild(n);
    return doc;
  }
  Node word = doc.createElement("word");
  Node keyword = doc.createElement("keyword");
  keyword.appendChild(doc.createTextNode("No match found");
  word.appendChild(keyword);
  doc.appendChild(word);
  return doc;
 }
```

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

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 also only temporary, when new words may be inserted the size of each index file grows and the 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 actual used) and concurrency control (transaction management) (see section 2.3.2).

The above introduction and evaluation of working with DOM (using a 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 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/tokenized, 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, when implementing updates application programmers are "on their own"

We also build the dictionary search function with SAX. Specifically the implementing callbacks 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 function `startElement`, `endElement` and `characters`, implemented to search for a keyword in a dictionary. Basically the examples maintains a state and takes appropriate action correspondingly. The example searches by comparing keyword with appropriate (characters that represent keywords in dictionary) characters read (lines 38-40),

```
 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 function `startElement`, `endElement` and `characters`. The example is simplified for illustration purposes, '..' denotes missing code. Full source code is found in appendix B.

when word is found a result is build using a StringBuffer (lines 13, 25, 42-43, 47).

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

SAX development is more challenging and less intuitive than DOM development, because the API does not provide functionality for accessing tree structured data.

SAX does not store document structure and content. If the document structure and content is needed while parsing the document, it must be kept by the application programmer. To build a document representation is a tedious process and most likely not related to the application logic tackled by the programmer. SAX is therefore not an intuitive choice if a tree-oriented representation is needed.

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. Because of it's design, SAX implementations is generally faster and requires 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 contains 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:apply-templates 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", that is any keyword element.

XSLT is typically used to transform XML documents, into an 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] although 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 suited, because XSLT can, as mentioned, 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 a XSLT program that selects and returns `word` elements which have a `keyword` with character data content equal to foo.

```
 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

The example illustrate pattern matching and template rules. Line 3 is a template that match the dictionary element (root element). Line 4 perform a pattern match on templates that match `word` elements with a `keyword` element with text content foo. Line 7 is a template that match anything and just return the matched element.

The main strength of XSLT is that it is 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], that is, 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 document, in some form, on some external storage device.

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

When dealing with persisted XML documents a number of approaches are used. XML documents can be stored in flat files described in section 2.3.1. Relational Database Systems may be used to store XML documents, 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 is introduced in section 2.3.3.

2.3.1 Flat files

Properly the simplest and most commonly used approach to persisting XML documents is simply 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. 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, that is, 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 bidirectional 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 XML document and it's mapping illustrated in figure 2.3

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

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

```
<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

are not to be confused with DOM objects, that models the structure of XML documents.[31]

Both mapping types are `bidirectional`. That is, 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 there for easy to map to relational databases. Data-centric documents are often contain repeated structured and 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 irregular structure and can be considered as real documents, that is some text with a logical structure e.g. the document containing this thesis.

Using relational databases for storing XML data is in general feasible and a good choice for data-centric documents and on the other hand a poor choice for document-centered documents. This, among others, because they do not preserve document order, processing instructions, etc.

In order to initialize the tables in the relation database, the logic structured of the persisted XML data 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 does not conform to the existing tables, the database tables needs to be expanded and reinitialized (using a new logic description). This results in inefficient execution times, and relational databases are thus only feasible to use, when the XML data structure is already known.

Another problem regarding the structure of XML data exists when XML documents contain mixed content[3]. Table-based mappings are not able to handle such an object-based mapping efficiently. The structure of the XML documents is thus required to be simple in relational databases.

In cases 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].

[3]Mixed content occurs when element nodes contain both sub elements and text data

A large number of null values waste space and a large number of tables will lead to slow read and write operations due to a 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 Enable Databases typically adds functionality to a relational database, which allows for retrieval and storage of XML data. This is achieved using the existing functionality for the underlying relational database and extensional bidirectional functionality for converting XML to relational data and back. As such XEDs are just an extension of relational databases and therefore lacks of the same advantages and short comings (see 2.3.2).

2.3.3 Native XML Databases

The most recent advance in database technologies relating 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. Queries involving XML data spread out in several documents have execution times worse than XEDs, since all the documents must be retrieved fully into memory, in order to find the XML data. NXDs are thus not a feasible solution for documents used in data-centric applications.

2.4 Summary

All of the described approaches have their shortcomings and advantages. A better solution for persisting XML documents should aim at combining the advantages and eliminate as many as possible of the shortcomings.

Persistences in flat files lacks of concurrency control in multi-user environments, transaction control, security etc. All these issues must be handled by the application programmer. Databases on the other hand handle these automatically.

The aim must therefore be to develop a simple API for persistence of XML documents. The API is to give a tree-oriented view of XML data, e.g. a DOM like API, but parsing/unparsing of whole documents should not be done in order to avoid high processing times and lack of memory for huge documents. The XML database being stored should be of arbitrary structure and DTDs or XML Schemas should not be necessary.

Furthermore, aspects such as concurrency control, transaction control, security etc. should be handled by the API. This should be transparent to application programmers in order for him/her to focus on the XML 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 name to IP mappings to ensure effectivty 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 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 an a value-oriented fashion is described in section 3.2. We then describe the value-oriented programming model in context 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 by inherited). Strings in Java are treated as immutable objects, that is all "modifying" methods will create a new String object, leaving the "original" object unchanged. For example `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 Sharing

These 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*, that is 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.

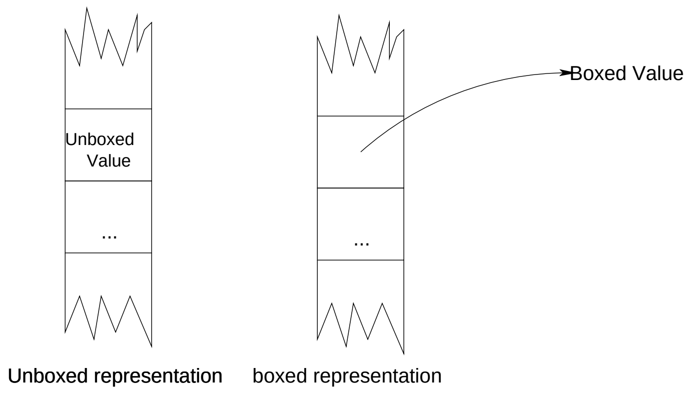


Figure 3.1: Unboxed and boxed representation of a value. In unboxed representation the value is saved directly in the allocated slot. In boxed representation the slot contains a reference to the actual value, which is saved else where.

References and the immutable nature of values enables 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 take benefit from this. This can be done by differentiating between references to immutable values and references to mutable data, i.e. reference to frequently updated 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, that is 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. Making Cells fixed size variables that can hold a reference to data, compromises for this.

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

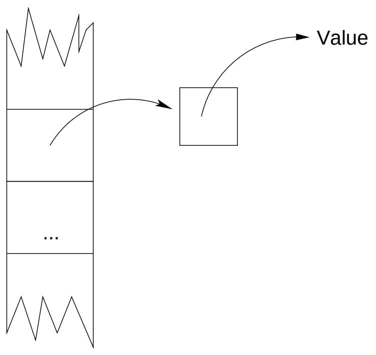


Figure 3.2: A cell representation of a value. The cell holds a references to a fixed sized slot. This slot holds a reference to the represented value. A cell is updated by writing a new reference (to the updated value) in the fixed sized slot.

3.2 Value-oriented trees

As XML documents are trees, as described in section 2.1, we describe how to work with trees in a value-oriented context. This section illustrates how to work with tree-structured data, using value-oriented programming. In value oriented programming all nodes (elements and leaf nodes) are considered values, and child references are value reference.

Consider again figure 3.3, 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, a tree turns into 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 arbitrarily number of parents.

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

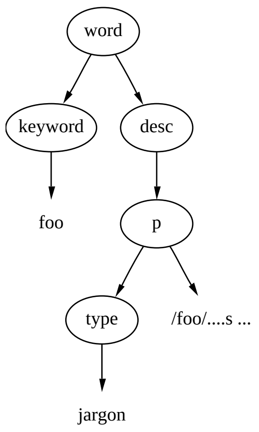


Figure 3.3: The left figure shows an example of XML data. The XML data have no attributes. The right figure shows the corresponding tree representation.

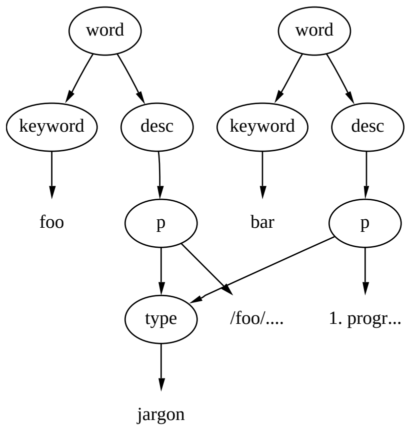


Figure 3.4: Sharing of subtrees, the element type is shared between the two trees

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

Consider figure 3.5, illustrating 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 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.

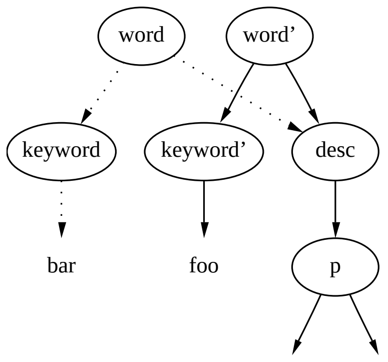


Figure 3.5: "modifying" subtrees mean creating new trees. When e.g. the `p` node is changed to `p'` (a new subtree is created), witch propagates all the way to the root node

3.3 Distributed concerns

This section describes the benefits of the value-oriented programming model, in a distributed environment. Before these benefits can be discussed, certain aspect of the programming model discussed above need to be addressed. The section above 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 the application programmer should as well make no distinction of the location of values, e.g. the location of values should be transparent to the application programmer.

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, 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 might be some other identifier. *Location indepent value references* are called *value references* and location dependent value references are called *locators*.

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

[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 refeference is used to retrieve a specific value. This is the process of unboxing a value.

must hold information of the value's location on the given machine. A *global 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. 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 them selves, they cannot be updated to referrer to a different value.

Location indenpendent 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, that is 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. That is 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].

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. An example of a simple reference resolver used in an in-memory environment. Here the reference resolver may use 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 benifits to an imperative (traditional) model.

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

Sharing of values the value model also 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 As all values are referred to by a reference and values them selves are never updated. Cell updates means updatings a cell with a value reference for the "new" value. Updating a value reference can be done atomic. It is possible to implement a simple light weight transaction control. As the "original" value never is changed, a rollback can be implemented simply by changing the reference back to the "original" value.

Chapter 4

The Document Value Model

The Document Object Model (DOM) and the Simple API for XML (SAX) described in sections 2.2.1 and 2.2.2 are widely used for accessing and modifying XML documents. Using these application programming interfaces have a range of shortcomings (described in the above mentioned sections).

The shortcomings of DOM are mainly a result of its imperative programming model. The value-oriented programming model described in chapter 3 solves these shortcomings. The *Document Value Model* adopts this programming model and the advantages, which it have compared to the imperative programming model.

The DVM is a high level tree-oriented application programming interface 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, entities in the Document Value Model is thus considered a value.

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

XML documents are represented by nodes. No interface distinction is made between element nodes and chardata nodes, that is Nodes represent either an XML element or XML character data. Nodes representing XML elements may have attributes and child nodes. Figure 4.1 illustrates a tree in the Document Value Model representing an XML document.

The API consists of several interfaces for representing the entities of a document and for specifying the functionality of the API. Any implementation of DVM must implement these interfaces.

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. An interface for creating nodes (XML documents), `XMLStoreFactory`, and an interface for persisting XML Documents (load and save), `XMLStore` are also specified.

The API only contain the most basic functionality for working with XML documents. Additional and more convenient functionality can be build using the basic functionality.

Section 4.1 describes how `Node` and interfaces related to this interface, are

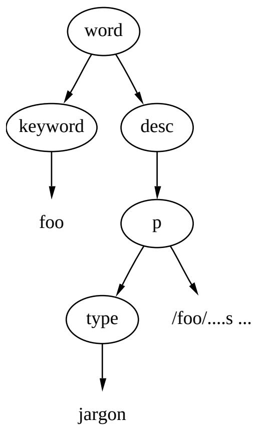


Figure 4.1: Graphical representation of the XML document from figure 3.3 in the DVM. Ellipses denote element nodes and simple text denote character data nodes. None of the element nodes contain attributes.

used to access documents. The following sections 4.2 and 4.3 describes the mutable nodes, and a how the visitor interface [10, p.331] are used to add functionality to nodes, without modifying their interfaces.

Section 4.4 describes the `XMLStoreFactory` interface for creating XML documents. Having described creation of documents section 4.5 explains how they can be modified, when represented by `Node`'s. The persistence functionality in `XMLStore` is described in section 4.6. Naming documents In section 4.8 is presented a utility library with additional functionality to the basic API.

4.1 Nodes - access to documents

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

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

The interface `ChildNodes` is used to represent the sub elements of an XML element. Attributes and their values are represented as character data using Java's `String` class. It is shown in figure 4.2.

Since 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*, which have a different meaning according to node type. In case the node type is `ELEMENT`, the node value is the tag name of the represented element. In case the type is `CHARDATA` the value is the character data.

Most of the methods in the interface have different meanings according to

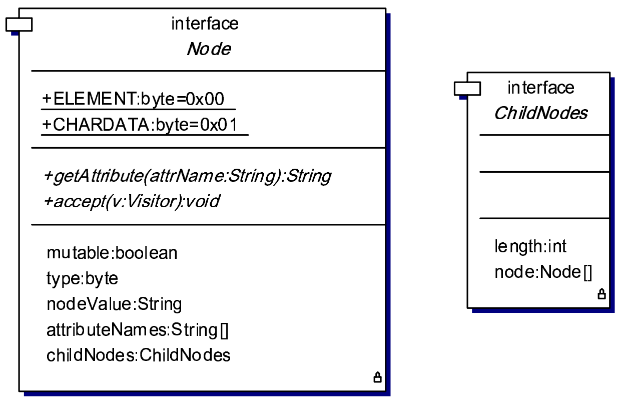


Figure 4.2: `Node` and `ChildNodes` interfaces.

the node type. 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, else false. (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 `ChildNodes` interface 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 (see fig 4.2) 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, see the HTML homepage [34]) The method is invoked on the `"word"` node and then call itself recursively on the children within the `"word"` XML document.

```
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 representing, which represents a 'word" element, to HTML.

The HTML code below is illustrates the `"word"` document representing `"foo"`. Tabs and line shifts are added in the illustration for the convenience of the reader.

```
<HTML>
  <BODY>
    <B>foo</B>
    <P>jargon</P>
    <P>/foo/ ...</P>
  </BODY>
</HTML>
```

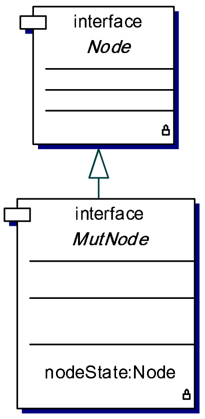


Figure 4.3: The `MutNode` interface.

The example illustrates how data in XML documents are accessed in the Document Value Model. Accessing data in value-oriented programming models is conceptually not any different from accessing data in the imperative 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 the application programmer to provide a more convenient and usable interface. The child nodes are obtained through the method `getChildNodes()` and the interface `ChildNodes`. Also if they must be loaded from storage before usage (loading is tansparant).

The interface `Node` is used for representing both XML elements and character data. The result is that `Node` methods have different meanings according to the node type, and that casing on the node type might be performed often. (The example shows casing by `switch(node.getType())...`). The solution would be to introduce of sub interfaces representing elements and character data. Such a solution may lead to excessive down casts, which are unsafe and slow.

4.2 Mutable nodes

In certain situations only programming with values can be cumbersome and inflexible, as described in section 3.1. The introduction of cells in the value-oriented programming model solved the problem.

Value references and thereby cells are not accessible when documents are traversed (see section 4.1). Cells can thus not be acquired for child nodes during traversal. As the `Node` interface does not allow modification (nodes are values) some other mean of representing cells in the Document Value Model is necessary.

The interface `MutNode` is introduced to represent cells. This interface represents *mutable nodes*, that is nodes, who's node value, attributes and children may be changed. As mutable nodes also represents values the `MutNode` interface inherits the `Node` interface. The interface is shown in figure 4.3.

Using mutable nodes the *state* of the node is updateable. 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.

Mutable nodes are specially 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 illustrates usage patterns of immutable nodes.

The dictionary example used through out the report can be extended to contain information of most popular words, i.e. words requested most often. In order to provide this information a counter can be attached to each word to keep track of how many count have been performed for that particular word. The structure of a word subtree is then modified as shown in figure 4.4 to contain a 'count' element node, which contains the 'request count' in a character data node.

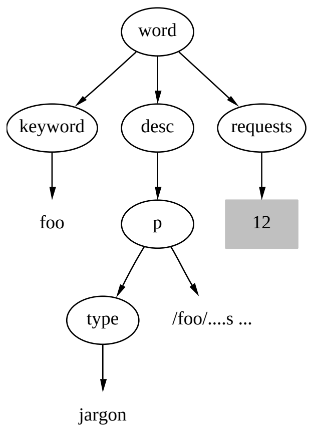


Figure 4.4: The 'word' nodes are modified to contain a search counter. The character data node, which contains the count is mutable. This example shows a subtree representing the word 'foo' which have been found 12 times.

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 updation of the search count. (The example is a slight modification of `XMLStoreDictionary`'s search functionality given in appendix B).

Notice that down casting is necessary to invoke the `setNodeState` method of

```
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 succeeds 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.

`MutNode`, which is expensive during performance. Since mutable nodes are not to used thoughtlessly, but only in special cases, this is considered an acceptable solution.

Invoking the methods `getX()` from the `Node` interface on mutable nodes should be done with care. Consider a case where the methods `getAttribute-Names()` 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 might occur. Access to the mutable nodes is thus safest through use of the method `getNodeState()`.

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

4.3 Adding functionality

Application programmers may need additional functionality when working with DVM. Using functional languages this can done elegantly with higher order functions.

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. The visitor pattern provides many of the advantages of higher order functions.

The Visitor pattern is introduced by the interface `Visitor` and the `accept` method in the `Node` interface. The `Visitor` interface is shown in figure 4.5.

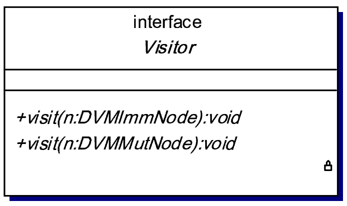


Figure 4.5:

The `accept` method is already described in section 4.1. The visit methods takes either an immutable node or mutable nodes as parameters. The code for visiting a node is written within these methods.

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 have the advantage that traversal code is only written once.
2. The traversal code is written by the application programmer in the `visit` methods. The advantage of this approach is greater flexibility, since the traversal can be stopped within the `visit` methods.

The Document Value Model adopts the latter approach. This due to the fact that document can be arbitrarily large and a whole traversal of such documents might not be desirable.

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` methods in this case contains similar functionality.

```
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. The value of these are simply written.

The visitor pattern allows for a flexible way of implementing additional functionality to be performed on document trees. By letting the application programmer implement the traversal code, the flexibility is even improved. This have the draw back though, that application programmers are to implement the often same traversal code in each `visit` method in all `Visitor` implementations.

4.4 Creation of documents

To create new XML documents and modify existing documents functionality for node creation is necessary. This functionality should not reveal the specific implementation of the Document Value Model (DVM).

The Abstract Factory pattern [10, p.87] is used for specifying methods for creating nodes, child nodes and instances of the `XMLStore` interface described in section 4.6. The interface can be seen from figure 4.6. An additional interface, `Attribute`, is introduced.

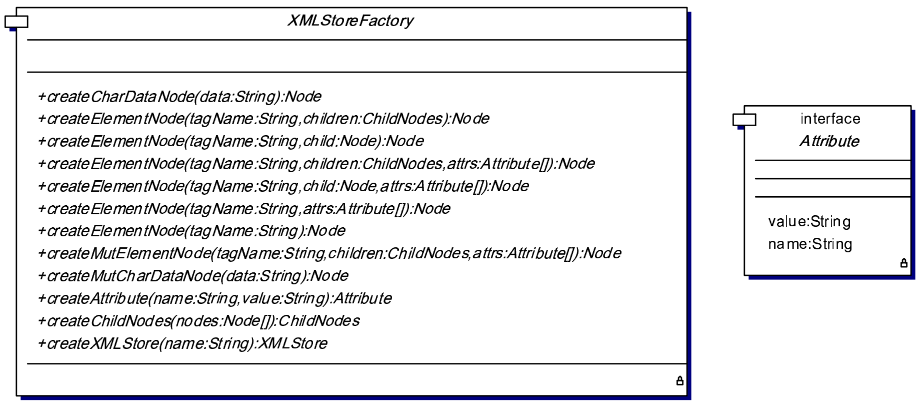

XMLStoreFactory
+createCharDataNode(data:String):Node
+createElementNode(tagName:String,children:ChildNodes):Node
+createElementNode(tagName:String,child:Node):Node
+createElementNode(tagName:String,children:ChildNodes,attrs:Attribute[]):Node
+createElementNode(tagName:String,child:Node,attrs:Attribute[]):Node
+createElementNode(tagName:String,attrs:Attribute[]):Node
+createElementNode(tagName:String):Node
+createMutElementNode(tagName:String,children:ChildNodes,attrs:Attribute[]):Node
+createMutCharDataNode(data:String):Node
+createAttribute(name:String,value:String):Attribute
+createChildNodes(nodes:Node[]):ChildNodes
+createXMLStore(name:String):XMLStore
interface
Attribute
value:String
name:String

Figure 4.6: `XMLStoreFactory` and `Attribute`.

By using the Abstract Factory pattern for creation, the code of application programmers will not become full of constructor calls for creating new instances. This provide the advantage that the implementation of the DVM interface can be exchanged with another implementation without affecting the existing DVM applications.

The create methods are described as:

`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 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 for the process of 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 the process of creating 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 or trees application programmers may use already existing nodes, and thereby share nodes. Nodes can be shared within the same document or between documents. As discussed in chapter 3 this can be done without concern to validity of data and therefore without implementing complex update protocols.

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

Shared nodes is located on disk or on another machine in the network. Using shared nodes versus non-shared nodes makes no difference to the application programmer.

```
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

As described in chapter 3, the difference in value-oriented and imperative programming style is the way data is manipulated.

Using the Document Value Model (DVM) 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 from the node being modified can be reused when creating the new node.

Example 4.6 illustrates how modification is carried out. The content of a "keyword" XML element 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 in DVM are done by building new documents. Example change keyword "bar" to "foo", in node "word". The example Continues code from example 4.4 and 4.5

As seen in the example this style of modifying new nodes (values) corresponds to the style described in section 3.2 for modifying value-oriented trees (directed acyclic graphs).

4.6 Persistence of documents

Working with XML documents implies loading and saving the documents. The Document Value Model (DOM) does not specify such functionality[1]. This task is left to the application programmer DOM.

When persisting XML documents the approaches often used by application programmers, is to store the XML documents in flat files or databases. Section 2.3.1 describes these approaches and their disadvantages.

Persistence of XML document are location and distribution transparent in the Document Value Model (DVM). The persistence functionality is provided by the `save` and `load` methods of the `XMLStore` interface, which can be seen from figure 4.7.

[1]The Document Value Model level 1 & 2 does not specify functionality for saving and loading. Level 3 specifies such, but is currently only a working draft

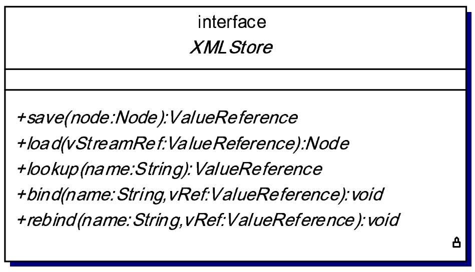

interface
XMLStore
+save(node:Node):ValueReference
+load(vStreamRef:ValueReference):Node
+lookup(name:String):ValueReference
+bind(name:String,vRef:ValueReference):void
+rebind(name:String,vRef:ValueReference):void

Figure 4.7: `XMLStore` interface.

`ValueReference` is an interface representing value references (defined in chapter 3). The interface is described in chapter 5.2.

The 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 required in section 3.1. The content of the value references is not accessible, not readable and not (re)producible. The application programmer can therefore not produce value references in order to retrieve documents from `XMLStore`.

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

```
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: The procedure of loading(, modifying) and saving a XML document.

The use of value references when saving and loading XML documents, makes the `XMLStore` interface provide a location and distribution transparent interface for loading and saving documents. Location of documents and protocols for distributed loading and saving is thus not of concern to the application programmer and full attention can therefore be paid to the manipulation of documents.

Further the application programmer is not affected by the movement of documents. This imposes that mobility of documents can be implemented in DVM with out any side effects on the application code.

Besides being location independent the interface does not reveal how documents are persisted, e.g. in flat files, in databases, in log structured storage. It only provides the application programmer with a tree view of the XML documents.

Another advantage of the `XMLStore` interface is that documents does 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. Instead the interface allows for lazy loading and thereby access and manipulation of arbitrarily sized documents.

4.7 Symbolic names for documents

To use the Document Value Model (DVM) to implement useful applications XML documents must be associated with human readable names rather than value references. Clients cannot share particular resources managed by a computer systems unless they can name them consistently. Thus, 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`. To share documents using the Document Value Model we need to provide a `name service`, which allows for retrieval of documents using human readable names.

Before describing a name service in the Document Value Model the concepts of `names` and a `name service` is defined in section 4.7.1. Then 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 (preferable 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 regarded pure names, which are not human readable. Locator are regarded addresses. Value references are as already described not adequate for retrieving and storing XML documents in the Document Value Model because 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 the methods `lookup`, `bind` and `rebind` shown in figure 4.7. 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`, that is looks up a value reference. If `name` does not exist `null` is returned.
`void rebind(String name, ValueReference ref)`	Updates a name-value references binding, i.e. after having invoked `rebind(name,valref)`, `name` is mapped to `valref`.

In the value-orientation programming model updates are performed atomically (see section 3.4). The `rebind` method thus provides atomic updates of the name-value reference binding.

Example 4.8 illustrates how an XML document is retrieved by 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"`.

```
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.

The name service makes it possible to retrieve XML documents by using human readable names in constrast 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. This is in contrast to the value-oriented model in which values are never removed.

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 thus required.

4.8 Utility library

The Document Value Model interface only provide the most basic functionality for accessing and manipulating XML documents. Writing applications using this Application Programming Interface (API) might become tedious and inconvenient.

A *utility library* containing extra functionality has been develop to improve the usefulness of the Document Value Model (DVM).

The need for additional functionality could be solved in two different approaches. The first approach is to extend the existing DVM interfaces with additional functionality. This has the shortcoming, that down casts are necessary to acquire access to the additional functionality. The second approach is the one adopted, that is write a utility library containing the additional functionality. This has two advantages: 1) functionality is accessible without down casts. 2) Existing interfaces can be used with new utility libraries.

The provided utility library has been implemented during development and testing of a DVM prototype implementation. Only a subset of the most important utilities are described. The functionality provided in 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.

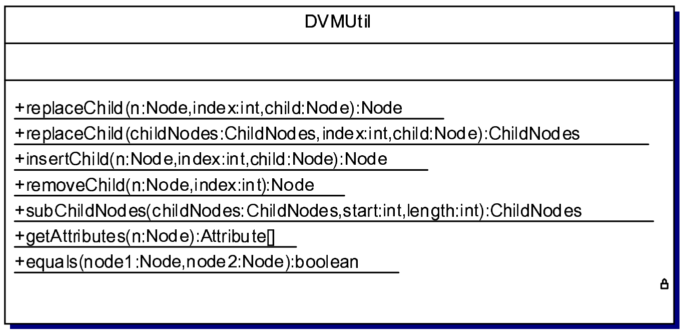

DVMUtil
+replaceChild(n:Node,index:int,child:Node):Node
+replaceChild(childNodes:ChildNodes,index:int,child:Node):ChildNodes
+insertChild(n:Node,index:int,child:Node):Node
+removeChild(n:Node,index:int):Node
+subChildNodes(childNodes:ChildNodes,start:int,length:int):ChildNodes
+getAttributes(n:Node):Attribute[]
+equals(node1:Node,node2:Node):boolean

Figure 4.8: DVMUtil interface

4.8.1 Access and modification

Access and modification of XML documents in the Document Value Model (DVM) might become inconvenient, since only the most basic functionality is provided. Specially the process of modifying nodes (by building new nodes) might become tedious and inconvenient.

The class `DVMUtil` contains functionality for easing the process of access and modification to XML documents. The class interface is shown in figure 4.8.

Method definitions:

`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)`	Return 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)`	Return 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)`	The method checks equality of two nodes. In order to be equal two nodes must have the same type. For character data nodes, the character data's (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 with in 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 insertion and removal 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. The difference from an imperative model, is that they do not modify nodes, but return new nodes

Example 4.9 purposely illustrates removal and insertion using utility methods `removeChild` and `insertChild`. For the functionality performed, i.e. replacement of a node, the `replaceChild` method is more convenient. 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 Value Model's interface. The semantics of the interfaces however differ significantly, as destructive updates is 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 of XML Store 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.

The functionality is part of the utility library and implemented in the class `DVMBuilder`. The class interface is shown in figure 4.9.

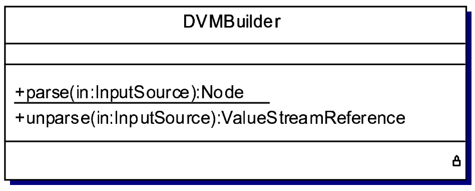

DVMBuilder
+parse(in:InputSource):Node
+unparse(in:InputSource):ValueStreamReference

Figure 4.9: The interface of the class `DVMBuilder`.

The methods of the interface 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 consists 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 specially useful in situations, where huge XML documents is persisted in flat files must be persisted in a DVM representation. (Chapters 6 and 7 describes such a cases).

4.8.3 Summery

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 the most basic functionality is specified.
- The DVM functionality is extensible, either by inheritance, development of utility libraries or by using the provided visitor pattern.
- DVM offer 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 using cells as described in chapter 3.
- The interface allows for convenient traversal of document trees, since references are not needed to retrieve child nodes.
- XML documents may be shared.

- Location and distribution of documents are transparent to the application programmer.
- DVM allows for lazy loading of documents (on request). This prevents loading whole documents into memory, when only a small selections are needed. The advantage is clearly that huge documents can build, without concerns of memory usage.
- In-memory nodes and on-disk nodes are treated equally by applications programmers.

The DVM also have shortcomings. These are not due the value-oriented programming model, but only to the interface design.

- The name service functionality does not allow multiple value reference to be bound to the same name.
- The name service functionality introduces an imperative aspect into DVM.

Chapter 5

XML Store architecture

An implementation of the Document Value Model (DVM) described in chapter 4 is a storage manager that handle persistence and distribution. This chapter discuss 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 referrers to a peer-2-peer network. The term *XML Store peer* referrers 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.

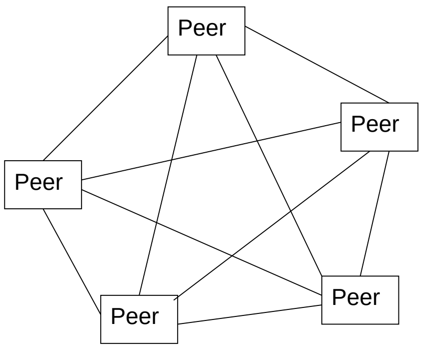


Figure 5.1: XML Store consists of XML Store peers. Each peer may communicate with all other peers in the network.

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 it's location in the network (ip and port) must be know. As described in section 3.3.1 value reference 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.

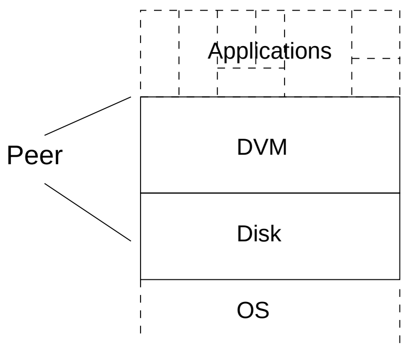


Figure 5.2: Single peer layered architecture consisting of a DVM layer and a Disk layer. Applications are build on top of the DVM layer.

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*, is the basic parts 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. These issues are 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 provided by the application programmer a value reference to the document is looked up using a name server

2. loading the document using this reference involves:

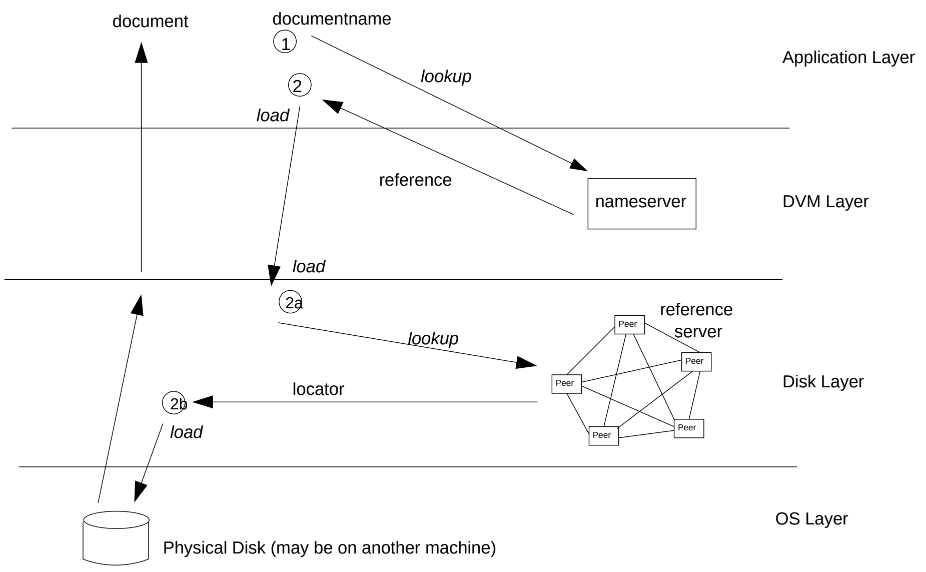


Figure 5.3: Retrieving a document from XML Store. The value reference associated with a document name must be looked up using a name server. To retrieve the document from the value a locator must be looked up using a reference server. Given a locator the actual document can be loaded. (*Italic* text 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 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 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 is 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 requires 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.
2. It may not be feasible to keep all mapping in dynamic memory, as the amount of key-data pairs can be large. In order not to take up to much dynamic memory a persistent solution is preferable.

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.

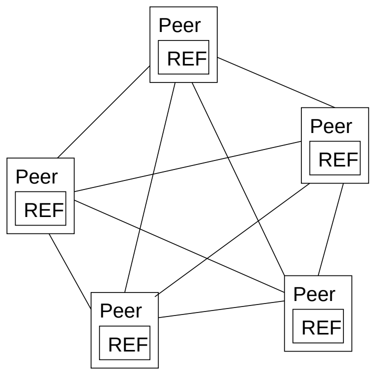


Figure 5.4: The XML Store peer-2-peer network. Each XML Store peer contains reference service functionality (denoted by REF). Communication between the peers takes place in order to provide a distributed reference service.

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 the basic functionality of a hash table (lookup, bind).

`Locator lookup( ValueReference vr )`	A locator is looked up given the value reference in the argument.
`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 the 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 does 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 the 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 a build 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 on 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. (IP packages contain the sender IP, which the peer extract and use to reply). 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 actions 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 to two data on two 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 will can resolve the value.

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

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 in 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 property contradicts one of the reasons for using a persistent mapping, i.e. the property that mappings should not take up to much memory (see the reference server requirements in the introduction of section 5.1). More sophisticated persistens mappings can be implemented using more advanced mapping techniques.

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 operation 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. That is, 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, and then "deleting a value" has no meaning. Further 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). The lack of support for deletion of values, result in that 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 a distributed loading degrades performance. Applying a proper strategy for caching distributed loaded values improves the 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 it 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 disk extensible as new features may be added simply by writing new "decorators". In the prototype implementation given with this report disk can be configured using a property file (see appendix A).

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 discuss some performance issues 5.2.4. The design and implementation of the disk API is described in chapter 6.

5.2.1 Locating values

Locating values on disk is necessary in order to retrieved values saved. 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

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

Figure 5.5: Simple Locator protocol

need any explicit knowledge of other peers because locators contain precise (and independent) route information to values. 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. A using disk that only uses locators have shortcomings. Such an architecture does not support mobility of values. If data are moved from one physical location to another 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. Two objections can be made against this approach. First, disks would not save in a log-structured manner. Second, when values are often moved, it would 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 enables mobility of values to be implemented, since the value reference will 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 exists 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 is also listed in section 3.3.1.

In the prototype implementation, given with this report, MD5 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 as simple implementation issue.

5.2.2 Storable values

So far, data that can be stored (and loaded) from disk have been called values. The sections describes what types of values may be stored on disk.

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 (on request). Value streams are compound values, consisting of either bytes (data) or references to other values (value references). As the disks must admit arbitrary sized input/output bytes are also returned in streams. Value streams can be expressed like.

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

where bytes is defined as

```
bytes = byte*
```

This regular expression shows that value streams consists of bytes followed by a sequence of value references and bytes, which is repeated zero or more times. Bytes 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[1], retrieving the first value from a given stream always returns the same value. Therefore values 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.

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

[1]Haskell is a lazy functional programming language

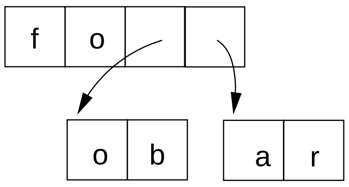


Figure 5.6: value streams and references

```
<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. If the disk interface alone should be used when working with XML, a lot of work would be left to the application programmer. (A more convenient API for persisting XML data is provided by the DVM layer, described in chapter 4).

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 supports 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. Cells do not hold values as they must be fixed sized, which value references are.

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 such caching schemes can easily be implemented as values are immutable. Cells do not offers 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 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 stream actually are larger than value reference (128 bit). Such a strategy would improve performance as fewer calls to reference server have to be made. This improvement is naturally aimed a XML documents containing many, but small nodes.

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). That is it should be build 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 of an imperative nature, which poses problems in a decentral name server architecture.

A value reference can not be updated to referrer to a different value, as they have an injective relation to they value they identify. A specific value references can thus be mapped to different locators on different reference server peers, as the locators all addresses the same value.

This is different to the name server. Names does not have an injective relation to the value reference they identify. A name is therefore only bound 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 look up using IP Multicast (as in the reference server solution). As IP Multicast is not reliable the look up might fail even though 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 a IP Multicast as when performing a lookup. Due to the missing reliability rebinds messages might not get delivered at peers, which is not satisfying.

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.

Further the name server is not used as often as the reference server. The bottle neck problem is thus non existing for small XML Store’s, that is XML Store’s 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 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 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

Applications access the XML Store solely through the *DVM core module*.

The DVM core module provides the functionality of the Document Value Model (DVM) interface. To provide this functionality the module makes use of 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 thereby to give 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.

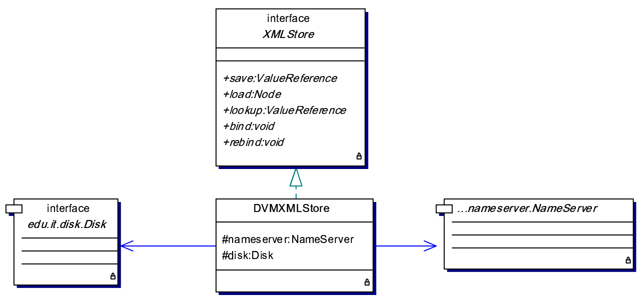


Figure 6.1: `DVMXMLStore`, the class implementing `XMLStore`.

XML documents in DVM are represented by the `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 the classes and their relations.

Document are loaded lazily. Such loading strategy can be implemented for immutable nodes. Section 6.1.1 describes how lazy loading is implemented. Ones having described lazy loading the procedure for saving nodes, can be described. This is done in section 6.1.2. Section 6.1.3 describes the implementation of mutable nodes. Finally section 6.1.4 describes the implementation of child nodes. This is done because the data structure used for keeping child nodes influence performance.

6.1.1 Loading documents

XML Store must be able to handle documents of arbitrary size. A loading strategy where document are loaded fully into memory is therefore not applicable. The applied loading strategy assures, that whole documents are not loaded

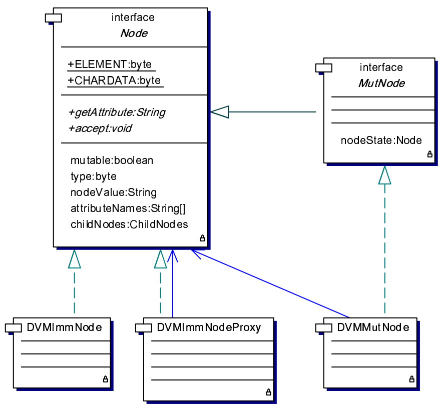


Figure 6.2: Classes implementing `Node` and `MutNode`.

into memory. Instead they are loaded lazily. That is 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 the data is requested (through use of the node interface).

Proxies initially do not contain node data. Instead the data is loaded when being accessed. In order to load the node data, the proxy instance contains 1)a value reference to the data which is used to load the data.

In short the process of loading a node is described as:

- 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` works accordingly:

`getType`	If not already loaded, the method loads the node type from disk and returns it. The type is kept in the proxy node for future method invocations.
`getNodeValue`	If not already loaded, the method loads the node value from disk and returns it. The value is kept in the proxy node for future method invocations.
`getAttribute`	If not already loaded, the method loads all the node attributes from disk and returns the relevant attribute value (if the attribute exists). The attributes are kept in the proxy node.
`getAttributeNames`	If not already loaded, the method loads all the attributes from disk and returns the attribute names. The attributes are kept in the proxy node.
`getChildNodes`	If not already loaded, the method loads information of all the child nodes and create proxy instances for the child nodes. This way the children are not fully loaded, but only represented by proxies.

Using this load strategy only necessary node data is loaded into memory. That child nodes are loaded lazily, means that child nodes not accessed is not loaded from disk. Huge parts of a document (tree) might therefore not be loaded from disk. This is illustrated in figure 6.3.

The above described lazy loading strategy is not enough for solving memory problems. The following section describes how filled memory usage is avoided by discarding data from memory.

Controlling memory usage

When an XML document is traversed the Document Value Model tree representation of the document grows in memory. This is a consequence of the lazy loading of child nodes. As more and more child nodes are loaded and their data as well, memory usage increases. For huge XML documents this imposes a problem, as traversing such document may 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 are again only contains 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 a simple and straight forward solution.

A short coming of the solution is the missing differentiation between popular and not popular data, i.e. data accessed frequently might be repeatedly discarded from memory and loaded into memory if the queue is full. The same data might therefore be loaded repeatedly.

A solution could be to implementation a strategy using a popularity degree of data. Data less frequently accessed (and therefore less popular) is discarded from memory before more popular data.

Summery

Using proxies representing persisted nodes allows for lazy loading of document data. The lazy loading is even provided transparently to the application programmer. That is if documents are loaded fully or lazily is not revealed by the `XMLStore` interface.

Controlling memory usage is a clear advantage to the application programmer. Application programmers do not have to consider documents sizes, when implementing applications for accessing XML documents.

The implemented strategy does not consider node sizes. Even in situations, where nodes represents a huge amount of persisted data, all data is loaded into disk. Two possible cases for such situations exist:

1. Element nodes with a huge number of children. All references are loaded and the same number of proxy objects are created. The dictionary document used through out the report and presented in chapter 2 represents such a case.
2. Node values may be huge (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 document parts within 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. For nodes representing character data, the character data is saved on disk. For nodes representing elements, the tag names, attributes and references to the child nodes are saved.

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.

Applying the recursive postorder traversal saving means that element children are saved before the element itself. This is necessary in order to produce the value references for the children. If nodes in a document, that is 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.

The functionality for saving documents is encapsulated in the class `SaveVisitor` implementing the `Visitor` interface.

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

That unparsing and saving is not performed for already saved nodes (represented by `DVMImmNodeProxy` or `DVMMutNode` instances) is a great advantage since whole subtrees (having the nodes as roots) are not saved unparsed again. This increases the performance when saving documents.

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. The chosen strategy thus enables sharing of nodes as described in section 3.2.

6.1.3 Mutable nodes

Sharing nodes within and between documents requires the state of mutable nodes to be kept up to date.

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 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 have been saved, `getNodeState` and `setNodeState` does not just return/change the node state kept in memory. The methods returns/changes the node state persisted on disk.

When `DVMMutNode` instances have been saved, they do not just contain a value reference like `DVMImmNodeProxy` instances do. They contain cell references. Changing a persisted nodes 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 for keeping child nodes in memory affects worst case access and insertion times.

Child nodes are kept in 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.

The draw back of arrays is time to add new elements. Time to add elements in arrays are O(n), as a new array have to be created an all elements must be copied, with the correct implementation this can be avoided.

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. Value 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, that is which basic operation 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 in is used by the DVM module to implement the 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 stream 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.

The operations for creating value streams are basic, however with these basic operations it is possible to create more complex (convenient) functions for creating value streams.

Using value streams

Value Stream offer a simple API for retrieving values from streams. Value Streams are as mentioned values them selves and acts differently than "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 lead 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. That is 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()`. Method skip enables skipping 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`. `RandomAccess-File` 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. which provides functionality for reading and writing anywhere in a given file.

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. That way writing becomes efficient as no time is spend seeking a place to write.

Loading

Loading values is done using a locator. Locators hold offset and length of values to be loaded. Loading of 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. These streams behave differently from imperative streams that remove values as they are read. Value streams are values themselves. Calling a method for retrieving a value will successively return the same value. 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 value 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). `SimpleValueStreams` are created using the create methods in the `ValueStream-Factory` (The factory only returns `SimpleValueStreams`). The class `DiskValue-Stream` reads from the random access file and the `GlobalValueStream` reads values from other XML Store peers in the system.

`DiskValueStream` and `GlobalValueStream` are built on top of regular imperative streams. To implement these two, classes that extend Java's `Input-`

`Stream`[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.

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 (the disk will be listening). When data is retrived 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 from using and `RAFInputStream` a 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. That is 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 have the interface represented by `ReferenceServer`. The `ReferenceServer` interface provides two methods `lookup` and `bind`, which were described in section 5.1.

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

Figure 6.7 illustrates the interfaces and classes and their relations.

[1]InputStream represents an input stream of bytes. The class interface is imperative.

[2]`BuffereredInputStream` adds buffer functionality to another input stream.

Classes implementing `Comunicator` must implement two methods:

`void send(byte[] msg)`	sends *msg* through the network. In case of the `MulticastCommunicator` *msg* is send to all subscribers of a specificed IP Multicast group, as described in section 5.1.
`int sendReceive(byte[] msg, byte[] respons )`	sends *msg* through the networks, waits for respons (which is put into *respons*) and returns the number of bytes in respons. The `MulticastCommunicator` sends the msg to all subscribers of a specifed IP Multicast group and wait for respons from any subscriber, when a respons is received all other responses are ignored.

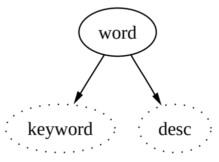
The reference service protocol implemented in `GlobalReferenceServer`, requires a distributed lookup. This is performed using the `sendReceive` method of the `Communicator` interface. The lookup request package have the structure:

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

`ACTION` is a byte, which denotes the requested functionality. In the lookup request package the byte denotes that a lookup is requested. `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 respons just includes the locator. The `Communicator` handles protocol information about which port the respons shuld be send 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).



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

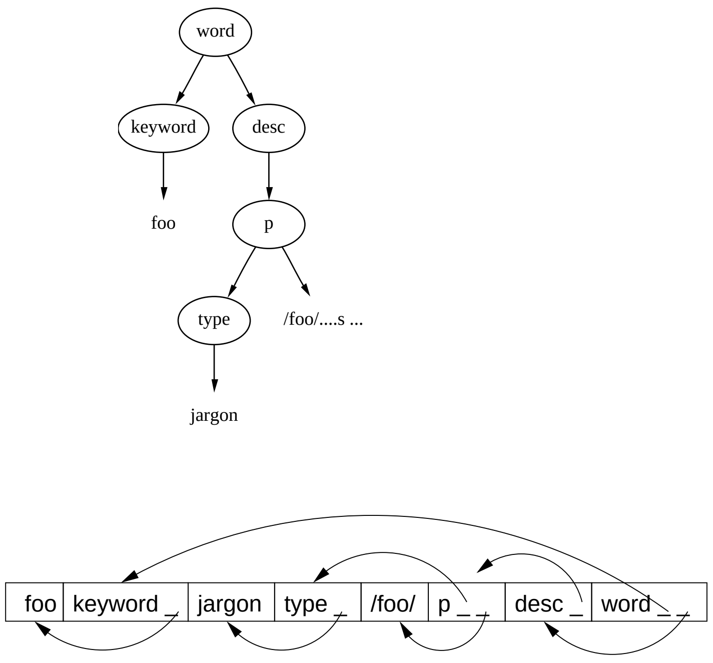


Figure 6.4: Unparsing nodes into value streams. Each node is unparsed and saved to disk. The bottom figure illustrates the resulting sequence of value streams (as they are saved to disk storage) when unparsing the XML document from the top figure. '—' separates valuestreams, '|' combined with arrows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

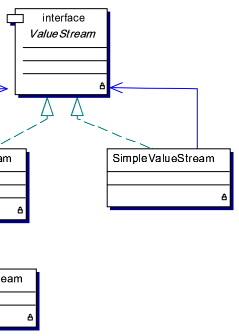


Figure 6.5: Class diagram of value streams.

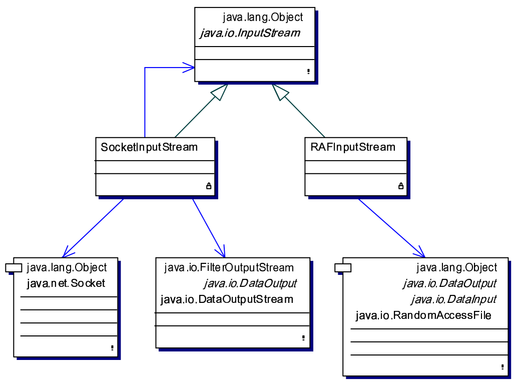


Figure 6.6: Class diagram of imperative streams

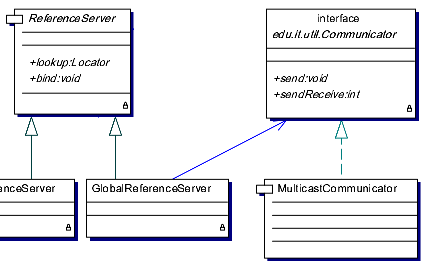


Figure 6.7: Class diagram of reference server

Chapter 7

Evaluation of the Document Value Model.

XML Store provides the Document Value Model (DVM) interface presented in chapter 4 for value-oriented storage, access and manipulation of Extensible Markup Language (XML) documents. The XML Store functionality must be usable and adequate for working with XML documents and should provide advantages compared to working with an imperative interface and imperative functionality.

The value-oriented interface and functionality of XML Store is evaluated through development of sample applications. Focus is on:

1. adequacy and usability, can applications be build easily and conveniently.
2. straightforward programming, can application programmers focus on application logic or must other (external) considerations, such as memory usage and concurrency, be considered.

The developed XML Store prototype implementation is used for building the sample applications. In order to evaluate the value-oriented XML Store functionality against imperative functionality sample applications has also been developed using the Document Object Model (DOM) interface.

Two sample application topics are presented, a small application that count nodes in an XML document, and the dictionary example used through out the report in its full length.

The XML Store is evaluated in section 7.1 through the node counter application. In section 7.2 evaluation is performed using the dictionary application. Section 7.3 provides a summery of the evaluation.

7.1 Node counter

Small applications with simple functionality must be easy to implement. XML Store's usability for such small application is evaluated through a simple sample application.

Node counter is an application, which count the number of nodes in an XML document, i.e. the number of XML element and XML character data.

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 node the method counts the number of nodes in the sub tree represented by the node. The result is produced recursively as follows:

- The root node of the subtree represents character data: 1 is returned.
- The node (root node or not) represents an XML element: The node count is produced by summarizing the node counts for the elements children. If a child represents character data, the element node count is incremented with 1. If a child represents an element the child's node count is produced by a recursive call of `count` and the current node count is increment with the count of the child node.

The node counter application must be started with two arguments. The second argument is the name of the XML document to count nodes in. The first argument is the name of the XML Store which the XML document is loaded from. The application can be seen below. To simplify the shown sample code, 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 application can be invoked from the command line through the call:

```
java -classpath xmlstore.jar NodeCounter myxmlstore mydoc
```

This example illustrates that a simple application, which count nodes, can easily 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 without using value references to obtain child nodes. Value references are thus only exposes when obtaining a document root node (as in lines 20-21). This allows for convenient traversal of documents.

The example provides an advantage with regards to disk I/O. The value of child nodes representing character is never loaded. This is due to the lines 7-8 and the lazy loading property of XML Store (see section 6.1.1). In line 7 the node type is accessed (and thereby loaded lazily). A call of `getType()` only enforce loading of the node type, i.e. the node value and possible attributes and children are not loaded. If the child node represent character data the node count is increased without invoking `count` on the child node (line 8). The character data is thereby never loaded and load time is saved.

The document name (`docName`) used in the application is '`mydoc`'. This name contains no information of location. When using the name to retrieve the root node in lines 20-21 no information of the location is revealed, because location independent value reference are used. The example thereby shows, the document location is not an issue, when using XML Store.

The location of nodes, i.e. in memory, on disk or on another peer is not an issue, when invoking `count`. The example shows that no special code is written to load nodes into memory (from disk or another XML Store peer) or to discard nodes from memory. That is the count method can be used to process any documents, with no regards to the document location. This is also a result of the lazy loading strategy.

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].

That is dictionaries are used to provide people with a joint understanding of words by providing information for each word.

A project group makes a good scenario for illustrating the usability of a specific dictionary. All members of a project group should have a joint understanding of the central terms used within the project. A dictionary containing the central terms is therefore ideal for providing the joint understanding of central terms.

The *dictionary application* is an application 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 data is accessible from any location and the data can be accessed by several applications.

That is the dictionary application should be build, 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. A user will thereby be able to start the dictionary application from the office computer, the home computer or any other computer.

Examples given through out 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 *key words*. It is worth recalling from appendix B, that word elements in the dictionary document are sorted according to their keywords lexicographically 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 does not violate the alphabetic order of the dictionary.
- it is not possible to insert a word already defined in the dictionary document.

The dictionary application has an interface, which contains methods for searching a keyword and inserting a new keyword (plus definition) into the dictionary document.

The XML Store is evaluated through implementing the dictionary sample application. In order to examine the advantages of a value-oriented programming style compared to an imperative programming style, the dictionary application is also implemented using the Document Object Model (DOM) interface.

Section 7.2.1 presents the dictionary implemented using the DOM interface. The following section, 7.2.2, presents the dictionary application implemented using XML Store.

7.2.1 DOMDictionary - an imperative dictionary application

Implementing the dictionary application using an imperative application programming interface, makes it possible to evaluate the value-oriented programming model introduced with XML Store against the imperative programming model.

The Document Object Model (DOM) is one of the most commonly used imperative application programming interfaces for accessing XML document data. Section 2.2.1 describes parts of a dictionary application implemented using DOM. The full dictionary application using DOM is presented within this section and used for a detailed evaluation of the imperative programming model and DOM.

DOMDictionary is a dictionary application implemented using the Document Object Model interface for searching keywords and inserting new keywords into the dictionary document. The dictionary document is a serialized XML document stored in a flat file.

DOMDictionary follows the dictionary functionality principles stated above (in the introduction to section 7.2). 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.

The methods `loadDict` and `saveDict` is used for saving and loading the 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. There after it invokes `binarySearch` to search the keyword. If the keyword exist 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 exist 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 sub packages are used to unparse XML documents in DOM representation to a serialized representation and write to output destination (standard output or a flat file). 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    }
```

```

  private boolean binarySearch( String word ){
    NodeList hws = dict.getElementsByTagName( "keyword" );

    int a = 0, b = hws.getLength() - 1;
    while( a <= b ){
      int i = (a+b) / 2;
      Node n = hws.item( i ).getFirstChild();
      String s = n.getNodeValue();
      int res = s.compareToIgnoreCase( word );
      if ( res > 0 ){ // greater, search right
        match = hws.item( i ).getParentNode();
        b = i - 1;
      }else if ( res < 0 ){ // smaller, search left
        match = hws.item( i ).getParentNode();
        a = i + 1;
      }else{ // found!
        match = hws.item( i ).getParentNode();
        return true;
      }
    }
    return false;
  }

  public Node keywordSearch( String keyword ){
    loadDict();
    Document doc = builder.newDocument();
    if( binarySearch(keyword) ){
      Node n = doc.importNode( match, true );
      doc.appendChild(n);
      return doc;
    }
    Node res = doc.createElement("word");
    Node kw = doc.createElement("keyword");
    doc.appendChild(res);
    res.appendChild(kw);
    kw.appendChild(doc.createTextNode("No match on keyword "
                                      + keyword));
    return doc;
  }

  public void insert( Node word ){
    NodeList nws =
      ((Element)word).getElementsByTagName( "keyword" );
    // only one keyword element exist within word
    Node child = nws.item(0).getFirstChild();
    String keyword = child.getNodeValue();

    loadDict();
    if( binarySearch(keyword) ){
      //allready exists throw exception
      throw new DictionaryException("Word exists!,
                                    "nothing inserted");
    }
    Node parent = match.getParentNode();
    Node newNode = dict.importNode(word, true);
    parent.insertBefore(newNode, match);

    saveDict();
  }

  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 below the dictionary document resides in the directory with relative path `dict` and have file name `foldoc.xml` The word searched for is `foo`.

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

Coding with the imperative DOM interface provides an at first glance acceptable solution to a dictionary application. The DOM interface is itself easy to use and paired with the Java package `javax.xml.transform` the DOMDictionary application seems a good choice for a dictionary application.

The DOM interface is a convenient interface for document (tree) traversal and retrieval of document data. Worth mentioning is the method `getElementsByTagName(String tagName)` which is invoked on a node. The method returns a list of all element ancestors with the tag name `tagName`. The method is used in lines 28 and 69.

The DOM interface and implementation never the less also contains a number of draw backs, which makes it unsuitable for implementing the dictionary application.

Parser configuration. The dictionary document has a size of 7.6Mb, when stored in a flat file. On the used PC's (on the IT-University) Crimson, Java's default XML parser, was not able to handle XML documents of such big sizes. To parse the document the Xerces parser [37] was used instead. This required a specific parser configuration within the DOMDictionary application. The parser configuration is made in lines 12-14. Choosing parser is an element of distraction removing attention from the application code.

Memory consumption. Commonly used DOM implementations parse XML documents into memory. This put a natural restriction on the sizes an XML document can have and therefore poses a problem when using the DOMDictionary application. If enough new words are added to the dictionary, the dictionary document will grow to a size not manageable in memory.

A solution for the application programmer could be to separate 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*'.

```
  private void loadDict(){
    // concurrency control needed
    // but not implemented
    String fileName = keyword.substring(0,1) +
                      "_" + dictName;
    dict = builder.parse( new File( dictName ) ) ;
  }
```

The DOMDictionary solution is not a straight forward solution anymore, since special care must be taken.

No sharing. XML data cannot be shared within or between XML documents. This because every DOM node (representing XML data) has one and only one parent node. That nodes have parent nodes are illustrated in lines 37, 39, 43 and 80.

Additionally every node belongs to a document node. The `importNode` method in lines 54 and 81 illustrates how nodes are imported into a document, and from then on belongs to the document.

Concurrency and transaction control. The imperative programming style is a destructive style, in which updates are made by changing data. When data is shared between several processes concurrency control is necessary, to ensure that data is not updated by processes, while other processes access or update the same data.

In imperative environment where several processes updates the same data, transaction control is needed. Using transaction control ensures that updates is not loosed. (With out transaction control processes can overwrite updates made by other processes - such a situation is illustrated in section 7.2.2).

DOM implementations (including the used one) does not provide concurrency control nor transaction control of files being loaded from and saved to storage. Such control must be implemented by the application programmer.

Concurrency control and transaction control is not implemented in the DOMDictionary, but should have been provided in methods `saveDict` and `loadDict`. The result is, that only one DOMDictionary application instance can be allowed to access and update the dictionary document.

Transaction and concurrency control can also be introduced by storing XML data in databases providing such controls instead of in flat files. For both solutions the simplicity of the application is lost, and implementing the DOMDictionary is not a straight forward job.

No location transparency. When loading the dictionary document the document location must be provided within the file name. That is if the dictionary document resides in a directory other than the Java Virtual Machine working directory, the absolute or relative path must be a prefix of the file name.

In the DOMDictionary the file name is provided as argument to the application invocation. The example given above illustrates how the location of the document is provided within the file name. Location transparency is thereby not provided by DOM implementations.

No distribution transparency. Distribution transparency is not provided by DOM implementations. It might be provided in some extend by the underlying file system, such that actual location of data (files) on intra nets is not revealed. E.g. the path `/import/stud/home/tejl/dictionary.xml` in a Unix file system does not reveal, which machine the file is located at. It may be a local machine or a network file system server.

If files are saved outside an intra net special code for distributed saving and loading of files are necessary when programming with the DOM interface. Such code is not provided in the DOMDictionary application since the simplicity and straight forward programming style would have been ruined. The DOMDictionary application is therefore only capable of accessing dictionary documents saved on an intra net.

7.2.2 XMLStoreDictionary - a value-oriented dictionary application

Advantages of XML Store and the value-oriented programming model introduced are illustrated through a sample application.

XMLStoreDictionary is a dictionary application implemented using XML Store for searching keywords and inserting new keywords into a dictionary document. The dictionary document is stored in the XML Store.

XMLStoreDictionary functionality is implemented, such that it fulfills the dictionary functionality presented in the introduction of section 7.2.

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 a 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 similar functionality to the equivalent `binarySearch` in the DOMDictionary application. It is used for both searching a keyword in the dictionary document and searching the position in the dictionary document where a new keyword should be inserted.

Recalling the dictionary documents structure, the root element is called dictionary. This dictionary element contains lots of word elements, which each contain a keyword.

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 contain 3 word elements sorted lexicographically after 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 after 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 exist 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 striped from imports of Java packages, exception handling and standard in-/output code in order to simplify the example.

```java
public class XMLStoreDictionary{
  private XMLStoreFactory factory;
  private XMLStore xmlstore;
  private Node dict = null;
  private ChildNodes words = null;
  private ValueReference ref = null;
  private DocumentBuilder builder = null;

  public XMLStoreDictionary (XMLStore xmlstore){
    this.xmlstore = xmlstore;
    this.factory = XMLStoreFactory.getInstance();
    this.ref = xmlstore.lookup("dictionary");
    dict = xmlstore.load(ref);
  }

  private int binaryLookup(String word){
    words = dict.getChildNodes();
    int lo = 0;
    int hi = words.getLength()-1;
    int mid = 0;

    while(lo <= hi) {
      mid = (lo+hi)/2;
      int compare = wordAtIndex(mid).compareToIgnoreCase(word);
      if(compare == 0) return mid;
      if(compare > 0) hi = mid -1;
      else lo = mid+1;
    }
    return (-lo-1);
  }

  private String wordAtIndex(int mid){
    Node keywordNode =
      words.getNode(mid).getChildNodes().getNode(0);
    return
      keywordNode.getChildNodes().getNode(0).getNodeValue();
  }

  public Node keywordSearch(String keyword){
    ValueReference ref = xmlstore.lookup("dictionary");
    if(!this.ref.equals(ref)){
      dict = xmlstore.load(ref);
    }
    Node word;
    int index = binaryLookup(keyword);
    if(index >= 0){
      word = dict.getChildNodes().getNode(index);
    }else{
      Node kw = DVMUtil.createElement("keyword",
                                     "No match on keyword " +
                                     keyword);
      word = factory.createElementNode("word", kw);
    }
    return word;
  }

  public void insert(Node word)throws DictionaryException{
    ValueReference ref = xmlstore.lookup("dictionary");
    if(!this.ref.equals(ref)){
      dict = xmlstore.load(ref);
    }

    Node kw = word.getChildNodes().getNode(0);
    String keyword = kw.getChildNodes().getNode(0).getNodeValue();

    int index = binaryLookup(keyword);
    if(index >= 0) {
     throw new DictionaryException("Word exists!, " +
                                  "nothing inserted");
    }

    index = -index-1;
    dict = DVMUtil.insertChild(dict, index, word);
    ref = xmlstore.save(dict);
    xmlstore.rebind("dictionary", ref);
  }
```

```
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). The application is invoked from the prompt with:

```
java -classpath xmlstore.jar XMLStoreDictionary
```

The above example illustrates advantages of programming with the value oriented model introduced in chapter 3 and the DVM interface presented in chapter 4.

Sharing. Due to the value-oriented interface and functionality of XML Store, XML 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 of nodes within the dictionary document exist. This is however most likely limited to sharing of simple nodes, such as `<link>...</link>`. An excessive amount of shared data within the dictionary document will thus not exist.

No parsing/unparsing. Lines 13, 42, 60 and 74 shows how the dictionary document is not parsed from and unparsed to a serialized representation when being loaded from and saved to storage. Time for parsing / unparsing documents is thus not necessary.

Memory consumption. XMLStoreDictionary is able to handle dictionary document files of any size. This is due to the fact, that whole documents may not reside in memory, but may reside partly in local storage and partly distributed on other XML Store peers. Memory consumption is therefore not a concern of application programmers using XML Store.

Location transparency. The lines 13, 42 and 60 showing loading and lines 74 showing saving of the dictionary document illustrates, how document root node location is transparent to the application programmer. The rest of the XMLStoreDictionary implementation no where reveals location of dictionary data. By using XML Store location transparency is thus achieved.

Not having to think of document location is a clear advantage in the XML-StoreDictionary. The application can be started at any physical location and used without providing location dependent document information.

This is also a general advantage when using XML Store. Data can thereby be moved from one physical location to another, without the concern of application programmers.

Distribution transparency. The dictionary root node may reside on another XML Store peer. Lines 12-13 shows how the root node is loaded without revealing if it resides locally or on another peer.

Parts of the dictionary document might reside on other XML Store peers. No were in the implementation of XMLStoreDictionary have been written code special for distributed environments.

This transparency of distribution is a clear advantage to application programmers, since applications becomes easier and more simple to write.

No concurrency control. `XMLStoreDictionary` does not contain any code for concurrency control. Due to the value oriented programming model where modifications are made by creating new documents, accessed document data is never updated/changed. Document can thus be accessed (read) by several processes without a need for concurrency control. Lines 73-74 to shows how modifications and saving are made, without any concurrency related code.

The name service functionality in XML Store introduces updateable values (i.e. imperative values), which identify value references. Those values are the names, used for naming documents. Updating the name to value references bindings in the name service introduces a possibility that updates might get lost. This is a shortcoming of the name service, which could be solved by implementing concurrency control.

A small scenario illustrates the shortcoming of the name service in a situation in which a dictionary modification is lost. Two users A and B each inserts a new word into the dictionary at the same time. The inserted words are different.

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 new version of the dictionary is 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 has thereby been “lost” to other users. This a clear disadvantage of the name server functionality. Unfortunately the name service interface does not allow an implementation of concurrency control. A proposal for modified name service functionality is presented in section 7.2.4.

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 thus 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 (i.e. a cell or mutable node) containing a value reference for the modified document, to contain the value reference for the old document. `XMLStoreDictionary` does not illustrate transaction control.

The Document Value Model interface of XML Store does however also introduce a shortcoming of the XMLStoreDictionary application (and other applications). The name service functionality of XML Store unfortunately introduces an imperative element into the value oriented programming style. The imperative element reside in the `rebind` method of the `XMLStore` interface. This method destructively **updates** the value reference bound to the dictionary document name. Using destructive methods in a multiple user environment requires concurrency and transaction control in order not to loose changes.

A small scenario illustrates a situation in which a dictionary modification is lost. Two users A and B each inserts a new word into the dictionary at the same time. The inserted words are different.

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 new version of the dictionary is 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 has thereby been “lost” to other users. This a clear disadvantage of the name server functionality and should be solved.

Besides the above problem with name service functionality the XMLStore-Dictionary functionality have been fulfilled by using XML Store. The basic Document Value Model interface has thus proved to be functional.

7.2.3 Dictionary Extension

The dictionary application is expanded to maintain a *search count* for each keyword in the dictionary document. The search count is the number of time a keyword has been searched with success.

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 have 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 to the found keyword is incremented by one and the dictionary document is unparsed and saved to disk. This is illustrated below.

```
  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", String.valueOf(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;
  }
```

The new implementation of the `keywordSearch` method in the DOMDictionary is coded in a straight forward manner. 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 is implemented by using mutable nodes. The `keywordSearch` method is updated to modify the mutable nodes on successful searches.

```
  public Node keywordSearch(String keyword){
    ValueReference ref = xmlstore.lookup("dictionary");
    if(!this.ref.equals(ref)){
      dict = xmlstore.load(ref);
    }
    Node word;
    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    }
```

This unparsing is required in the `keywordSearch` of `DOMDictionary` is in contrast to the modified `keywordSearch` method within the XMLStoreDictionary. 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.

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.

As using mutable nodes is equivalent to using cells simple transaction control can be implemented. The dictionary extension pedoes not illustrate simple transaction control.

7.2.4 Proposal for name service improvements

The evaluation of the Document Value Model in section 7.2.2 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 the likelihood of two processes' accessing the same values are low. Transactions are therefore allowed to proceed as if no problems exist. If conflicts arise one or more transactions are aborted and must restarted. The drawback of optimistic locking is the risk of starvation, where a process repeatedly have its transaction aborted and restarts.

Optimistic locking of values in the 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 is 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 a `ConcurrencyException` is thrown
`void rebind(String name, ValueReference ref, ValueReference oldref)`	Updates a name-value references binding, i.e. after having invoked `rebind(name,valref, oldvalref)`, `name` is no longer mapped to `oldvalref`, by instead mapped to `valref`. If updating is not possible (`oldvalref` does not equal the actual reference bound to `name`), then an `ConcurrencyException` is thrown.

In the central name server solution the consistency check (in case of `rebind` the equality of `oldref` and the actual value reference bound to `name` is performed at the central server.

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

```
public void insert(Node word)throws DictionaryException{
  ValueReference ref = xmlstore.lookup("dictionary");
  if(!this.ref.equals(ref)){
    dict = xmlstore.load(ref);
  }

  while(true){
    Node kw = word.getChildNodes().getNode(0);
    String keyword =
      kw.getChildNodes().getNode(0).getNodeValue();

    int index = binaryLookup(keyword);
    if(index >= 0) {
     throw new DictionaryException("Word exists!,
                                   nothing inserted");
    }

    index = -index-1;
    dict = DVMUtil.insertChild(dict, index, word);
    ValueReference newRef = xmlstore.save(dict);
    try{
      xmlstore.rebind("dictionary", newRef, ref);
      return;
    }catch(ConcurrencyException e){
      ref = xmlstore.lookup("dictionary");
      dict = xmlstore.load(ref);
    }
  }
}
```

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

7.2.5 Summery

A dictionary application has been implemented by using both an implementation of the Document Object Model (DOM) interface and by using the XML Store implementation of the Document Value Model (DVM) interface. This resulted in respectively the DOMDictionary application and the XMLStoreDictionary application.

Both the DOMDictionary application and the XMLStoreDictionary has been implemented in the most straight forward way. As consequence of this and the similar document structure, which the two interfaces provide, the implementations have similar structures.

The functionality of the two applications are very different though. The implementations of the DOMDictionary application showed that the DOM interface does not allow for a straight forward and simple implementation of the dictionary application. It was required for the application programmer to implement concurrency control, transaction control and any distributed aspects or to solve such issues by using database solutions for storing the dictionary data.

This were in contrast to the DVM interface of XML Store, which only showed a flaw in the name server functionality. This flaw is serious enough though. A simple correction of the name server functionality presented in 7.2.4 solves the flaw.

Properties of the DOM interface and DVM interface of XML Store is revised in the table below.

	DVM	DOM
Process arbitrarily large documents	+	-
Sharing of XML data	+	-
No parsing/unparsing before processing	+	-
Convenient interface which reflects semistructured data	+	+
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.

A difference between the two interfaces not shown in the above table, is their functionality. The DOM interface contains more functionality than the DVM interface and is thus more convenient to use. The methods `getElementsByTagName` and `getFirstChild` exemplifies this. The functionality of the DVM interface could have been extended with such and similar methods, but focus has been on developing an interface only providing the basic functionality.

The dictionary applications extended with search counters 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. It is thereby possible to update a single nodes without unparsing and saving the whole document as in DOM.

7.3 Evaluation summery

The node counter sample application and the dictionary applications DOMDictionary and XMLStoreDictionary have been implemented in order to evaluate the Document Value Model (DVM) interface of XML Store.

Implementing the node counter application illustrated that small simple applications can easily be build using XML Store. The node counter example didn't illustrate value-oriented aspect though.

These were illustrated in the XMLStoreDictionary, a dictionary application implemented by using XML Store. A dictionary application, DOMDictionary, were implemented by using the Domain object Model interface. The implementation of the dictionary application illustrated that in opposition to DOM, the XML Store provides sharing of XML data, distribution transparency and location

transparency. Memory consumption is of no concern to the application programmer and concurrency control is not necessary on simple load and save functionality. The simple name service functionality does unfortunately solve simple concurrency issues. A solution solving the problem has been presented in section 7.2.4.

Applications better suited for illustrating value-oriented advantages could have been chosen. The node counter does not illustrate sharing of values, and the amount of sharing in the dictionary application does not illustrate, that sharing is an advantage.

Chapter 8

Experimental results

The XML Store prototype implementation described in chapter 6 provides the functionality of the Document Value Model (DVM) interface described in chapter 4. The implementation must perform this functionality with a satisfying performance in order to be usable.

The performance of the XML Store prototype implementation has been examined through experimental measurements. The focus has been on time consumption, i.e. how fast is the different functionalities of the prototype implementation performed.

The performance of the prototype implementation have only been tested, for functionality central to XML Store usage. Such functionality is initializing XML Store and saving, modifying, retrieving and accessing documents.

Section 8.1 tests the performance of initializing new XML Store's. The performance of retrieving a document from an XML Store is examined in section 8.2. In section 8.3 the performance for accessing document content is examined. Section 8.4 describes save performance and section 8.5 describes the performance when modifying a document.

8.1 Cold start

Retrieval and persistence of documents using an XML Store requires the XML Store to be initialized. Initialization time is not only influenced by the instantiation of the XML Store, but also by existing files used for the initialization.

The *cold start* time of an XML Store is its initialization time.

During the execution time of a java virtual machine (JVM) XML Store's 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.

Existing disk and reference server files are used when initializing an XML Store. These files influence on the cold start are examined.

Section 8.1.1 examines the disk files influence on cold start and section 8.1.2 examines the reference server files influence.

8.1.1 Disk initialization test

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

The disk initialization test examines the influence of the disk file's size on cold start performance by measuring the initialization time of XML Store's using disk files of different sizes.

In order to examine the influence of the disk files size on cold start performance the influence of the reference server file must be neglected. As described in the following section (8.1.2), the reference server initialization is affected by the number of value reference to locator mappings in the reference server file. By keeping the number of mappings in the reference server file constant for all disk file tests, the reference server files contributes with a constant factor to the time measurements.

The disk initialization test is carried out as follows:

1. 10 XML Store's with increasing disk file sizes, but equal number of mappings in the reference server files is created.
2. Using the created store files and reference server files the cold start time is measured for each of the 10 XML Store's.

The test confirmed that the disk file does not affect the cold start performance. This is illustrated in figure 8.1, where it is seen that the cold start time is more or less constant for increasing disk size.

The disk file is not parsed during initialization and the disk file size therefore has no influence on the initialization. This is confirmed by the test.

8.1.2 Reference server initialization test

The number of value reference to locator mappings persisted in the reference server file are expected to influence the cold start performance.

The reference server initialization test examines the impact of persisted value reference to locator mappings on cold start performance. 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 Store's, all with the same disk file size, but increasing reference server file sizes (increasing number of value reference to locator mappings) were created.
2. The cold start time was measured for each of the 10 XML Store's using the created disk and reference server files.

The unambiguous result of the test showed (as expected), that the number of value reference to locator mappings greatly affect the cold start performance. When the number of mappings increases, the cold start performance time increases. The relation between the number of mappings and the cold start performance time is linear. Figure 8.2 illustrates the test results.

During the reference server initialization the existing reference server file (reference server) is parsed. The more mappings in the file, i.e. the bigger the file, the longer parse time. This is confirmed by the test result.

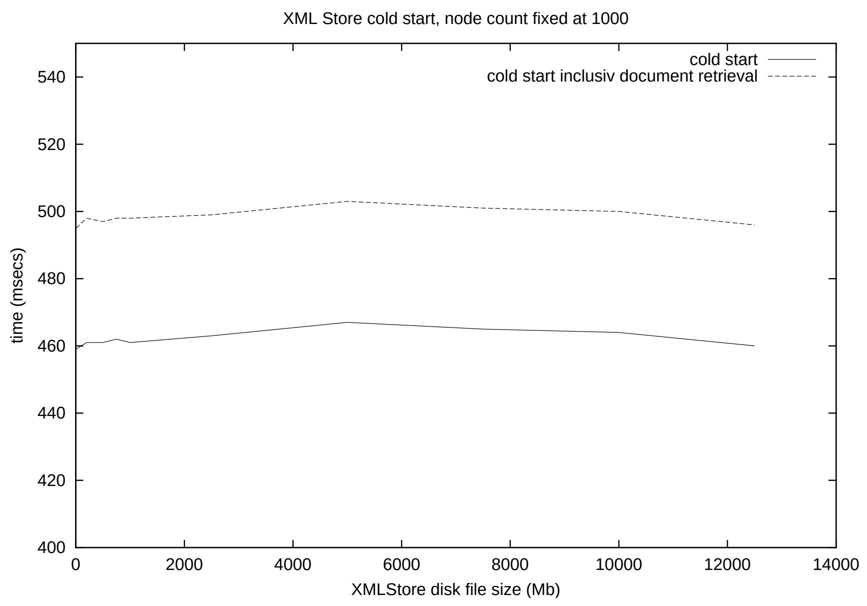


Figure 8.1: Cold start measures: XML Store's with a 1000 nodes, but increasing disk file sizes were produced. With the same amount of nodes, the XML Store's also contained the same amount of mappings. The cold start time was measured as a function of disk file size. The time measures were produced from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

8.1.3 Summery

The influence of a disk file and a reference server file on XML Store cold start performance have been examined. The test results clearly shows that the disk files size is of no influence on cold start performance, whereas the reference server file has influence on cold start performance. The cold start time increases with increasing number of value reference to locator mappings in the reference server file.

This is not satisfying, as XML Store's with a huge number of mappings will have a low performance on cold start. A different strategy for loading the reference server mappings is therefore needed.

8.2 Document retrieval

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

Retrieving an Extensible Markup Language (XML) document is retrieving the root node of the document. Performance is evaluated by measuring the retrieval time of documents.

The performance differences of document retrieval in XML Store and of-

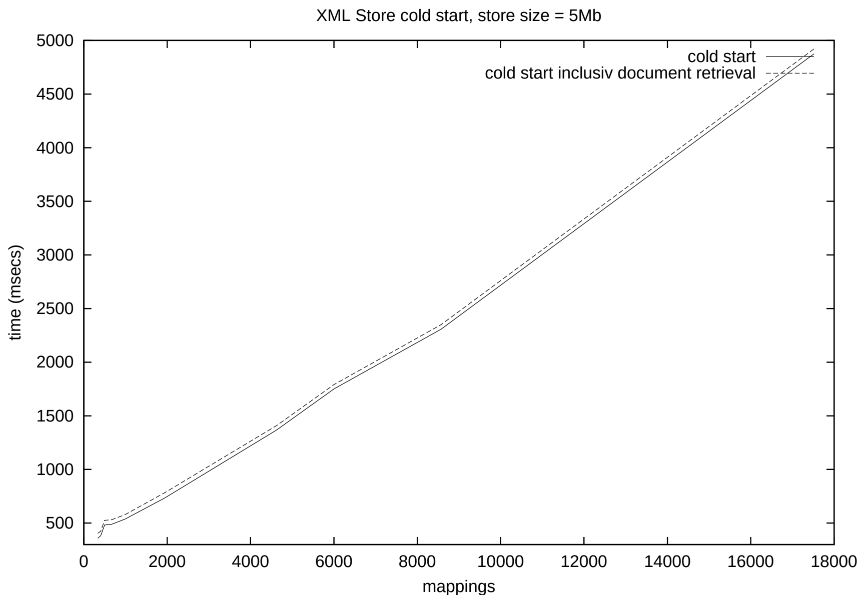


Figure 8.2: Cold start measures: XML Store's with a disk file size of 5 mega bytes, but different number of mappings were produced. The cold start time was measured as a function of the number of mappings. The time measures were produced from an average of 10 test runs.

ten used DOM implementations are illustrated by evaluating performance of document retrieval in both.

The performance behavior of document retrieval using XML Store is tested in section 8.2.1. The performance behavior for a typically used DOM implementation is evaluated in section 8.2.2. Section 8.2.3 summarizes the test results.

8.2.1 XML Store

Loading an XML document from XML Store is done by a lazy loading strategy, where XML elements and character data are loaded when needed. The document retrieval performance is greatly affected by this strategy.

The document retrieval performance 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. 2) Use the value reference to retrieve the document root node.

The test was carried by testing document retrieval on different documents. The test documents were reused from the cold start test (see section 8.1). The XML Store's used in the cold start test all consists of one single document. Document retrieval is therefore made by retrieving the single root node in the XML Store.

The test results showed that document retrieval was not dependent on document size nor the number of nodes in the document. This is illustrated by the figures 8.1 and 8.2.

That document retrieval performance is not influenced by document size nor node count 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 performance is unaffected by document content.

8.2.2 Document Object Model implementations

The upstart properties of the XML Store implementation are in contrast to the upstart properties of the most used Document Object Model (DOM) implementations. The performance for document retrieval therefore behaves differently in DOM implementations than in XML Store.

A test of the upstart performance of DOM implementations is therefore not comparable to the cold start tests of the XML Store. But testing the upstart performance in a DOM implementation will illustrate its different properties compared to the XML Store properties.

The upstart of the Document Object Model implementation is the parsing (or loading) of an Extensible Markup Language (XML) document. This test examines upstart times of a DOM implementation, i.e. the parse times of XML documents.

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.

In the test parse times for two groups of XML documents is measured. The first group of XML documents have a fixed number of nodes, while their *size* varies. The size of an XML document is the disk space it uses in serialized format. The second number of XML documents have a fixed size, while the number of nodes varies.

A test is performed twice, once for each group of XML documents. Each test is made by producing the retrieval time for each of 10 XML documents (of the group).

The test result for the group of XML documents with fixed node count shows, that the upstart time of the used DOM implementation increases as the document size increases. The relation between document size and upstart time is linear. The test results are illustrated in figure 8.3.

As the DOM implementation parses the whole XML document into memory, the document retrieval time should increase with increasing document size and the result is therefore not surprising.

The test result for the group of XML document with fixed size, shows a tendency, that the upstart time of the used DOM implementation increases as the node count increases. The upstart time also looks like it becomes asymptotically linear for increasing node counts.

The results also show though, that for small node counts, the upstart time is decreasing. The documents with a small amount of nodes is created by producing greater data nodes with big amount of data. A possible explanation for the decreasing upstart time, that the DOM implementation is slow in producing such DOM nodes, that is nodes with large CDATA sections. The results are illustrated in figure 8.4.

The asymptotically linearly increasing upstart time with increasing node count can be explained by the nature of XML parsers. For each node in the document a call back method in the parser is invoked (see section 2.2.2 for a description of call back methods). As invocation of a method takes time, the upstart time will increase with the number of call back method invocations, and thereby with the number of nodes in the XML document. This explains the asymptotically linear increasing upstart time.

The test results thereby 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 Summery

The document retrieval performance using the XML Store is not affected in any away by the document size of number of nodes in the document. This is in contrast to the performance for document retrieval using a DOM implementation. Here the document retrieval time increases both for increasing document size and for increasing number of nodes.

8.3 Document access

Retrieval of XML documents from the XML Store load of whole documents. Instead documents are loaded lazily (as needed). This affects performance when accessing documents.

Document access using XML Store is retrieving node type, value, attributes and children. 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 completely on the actual peer (the peer on which the keyword search is invoked) 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 the actual peer and an XML dictionary document residing on several peers.

Section 8.3.1 evaluates document access performance, when the XML dictionary resides on the actual peer. Section 8.3.2 evaluates the document access performance for an XML dictionary residing on several peers. Section 8.3.3 gives a short summery of the evaluations.

8.3.1 Local access

Document access performance depends 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, i.e. on the actual peer, is evaluated by measuring the time it takes to access data in a document stored locally.

The dictionary application presented in section 7.2 is used for the performance testing. 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. (During a keyword search the dictionary application will access data of the dictionary document). The keyword search is performed 10 times for the same keyword (the keyword is 'foo' and exists in the dictionary).

The test result clearly shows that the search time (or access time) is high for the first search, but low for the following searches. This can be seen from table 8.1.

search #	Time
1	6232
2	2
3	5
4	2
5	2
6	2
7	2
8	16
9	3
10	4

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

That the search time is much longer for the first keyword search is explained by the lazy loading strategy used in the XML Store implementation. During the keyword search in the dictionary lots of nodes are accessed. These nodes are all loaded lazily, i.e. when being accessed the first time. Since they are all accessed the first time during the first search, the access times and thereby search time is a lot higher for the first search compared to the following searches.

This shows that access performance in the XML Store is low, when nodes is being accessed the first time. In the prototype implementation nodes are loaded on request, i.e. one node at a time. A possible improvement would be to perform load buffering, thereby loading several nodes at a time.

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. The access performance of documents stored on other peers is therefore expected to be lower compared to documents stored on the actual peer.

The access performance of document stored on other peers, i.e. not on the actual peer, is evaluated by measuring the time for accessing such documents data.

As in section 8.3.1 the dictionary application presented in section 7.2 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. (The keyword is again 'foo'). All XML Store peers resides on the same intranet.

The test result again shows, that the first search is 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 loading the document on from the actual peers storage. The test result is shown in table 8.2.

search #	Time
1	57807
2	4
3	3
4	2
5	5
6	3
7	4
8	18
9	4
10	2

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

Just as when performing a search on a dictionary document stored on the actual peer, the search time is highest for the first search. This time the first search only requires much more time, because the dictionary document is loaded from other XML Store peers (see table 8.1 and table 8.2).

The relatively high loading time can not be considered satisfying. The solution to the problem might (as in section 8.3.1) be to implement techniques such as buffered loading and in-lining (see section 5.2.4).

8.3.3 Summery

Documents are loaded lazily, when document data are being accessed. The access performance therefore depends on load times.

The load time differs, when loading documents from the actual peer and loading distributed from other peers. When loading from the actual peer, the access performance is low but acceptable. When loading the document data from other XML Store peers the access performance becomes very low. The load performance might be improved by in-lining and load buffering.

8.4 Saving Documents

Saving XML documents is central to working with XML Store. Save performance is therefore a central issue in the XML Store performance.

The save performance of the XML Store prototype implementation is evaluated by measuring the times needed for saving documents.

In the prototype implementation document save times are expected to depend on the amount of data written to disk, i.e. 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 (a property of log-structured disk access - see The Design and Implementation of a Log-Structured File System [35]).

Section 8.4.1 tests how save times are dependent of document sizes. Section 8.4.2 tests the save performance with regards to the amount of already save data. Section 8.4.3 provides a summery of the save results.

8.4.1 Save functionality

The prototype implementation of XML Store provides simple save functionality with no buffering, asynchronous saving or the like. The performance of such simple functionality is most likely low.

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, and the time needed for saving is measured.

The test results shows, that save times increase with increasing XML document size. The relation between save time and document size is somewhat linear. The results are illustrated by figure 8.5.

When saving values on request like in simple save functionality of the prototype implementation, the save time must be expected to increase for increasing document sizes.

This is not satisfying for huge XML documents. A solution would be to introduce buffered and asynchronous save functionality.

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 (ref). The disk access time is therefore to be constant. The save time should therefore only be influenced by the amount of saved data.

The disk save performance is tested by measuring the save time of the same XML document in XML Store's with different disk file sizes.

The test is performed by creating 10 XML Store's with different disk file sizes. The same XML document is saved in each of the XML Store's and the save time is measured.

The test results shows, that the save time increases with increasing disk file size. This is illustrated in figure 8.6.

This result is in contradiction to the theory of log-structured save behavior, in which save times should only be affected by the amount of saved data.

From the figure is also seen that the increase in save time from saving in an empty XML Store to saving in an XML Store with disk file size of 12.5 Mb is only approximately 200 Msecs. This increase is not tremendous and certainly not a disaster.

The reason for the test result is not clear, but a possible explanation might be found in the use of Java's `RandomAccessFile` for writing data to disk. All though log-structured save behavior can be simulated using this class interface, the save performance achieved by using the class, might be affected by the amount of already saved data. If strict log-structured behavior is needed another implementation strategy might be needed.

8.4.3 Summery

The XML Store save performance is affected by the saved document sizes, such that save times increase with increasing document size. Such save performance can be improved by introducing asynchronous save functionality.

The save performance is however also affected by the amount of data already saved on disk.

8.5 Document modification

The value oriented programming style provides the XML Store with a performance advantage when modifying documents compared to modifying documents in an imperative programming style.

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.

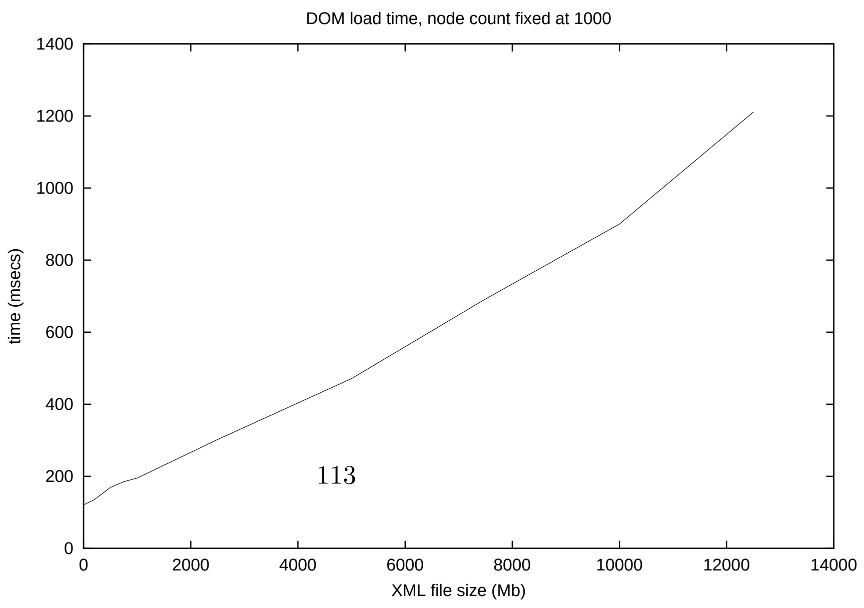
Document Size	Save Time	M Insert Time	M Save Time	Sum
5 Mb	132 Msec	19 Msec	3 Msec	22 Msec

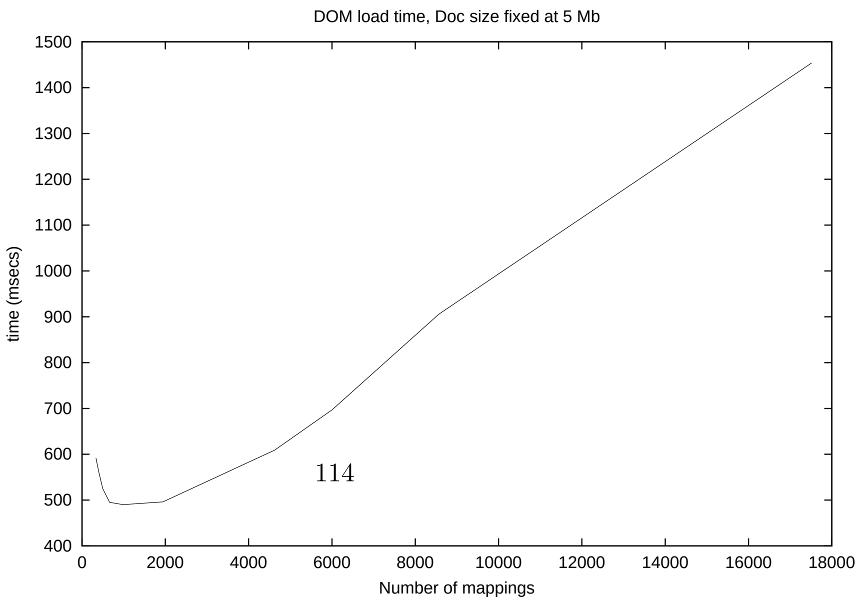
Table 8.3: Modification of an XML document. The second column contains the time for saving the whole XML document. The third to the fifth columns contains the modification times, first the time to insert new XML data, secondly the time to save the modified document and third the sum of the two.

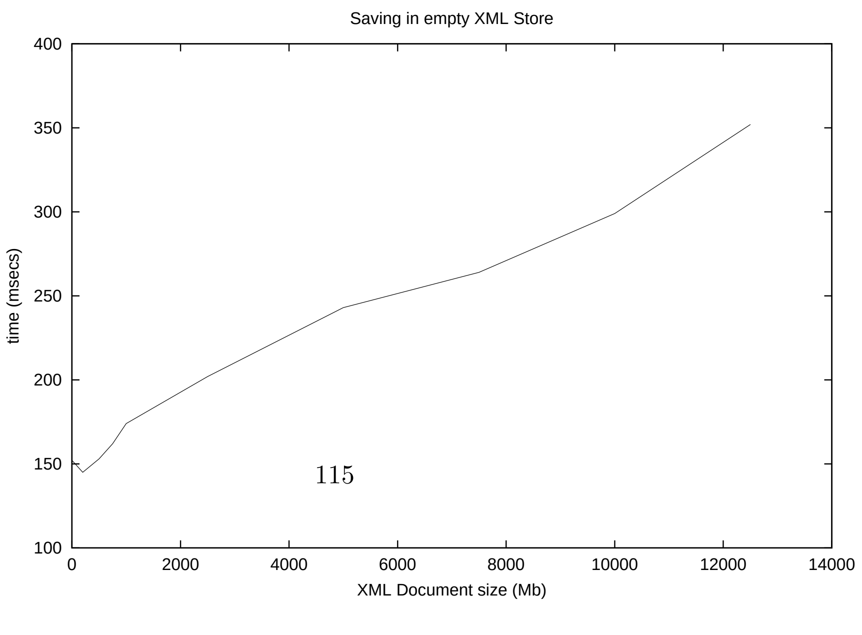
The test results shows how document modification in XML Store is faster than saving the whole document. This is a result of the share-create style used in value oriented programming (see section 3.2). The not modified document parts is thereby not saved again.

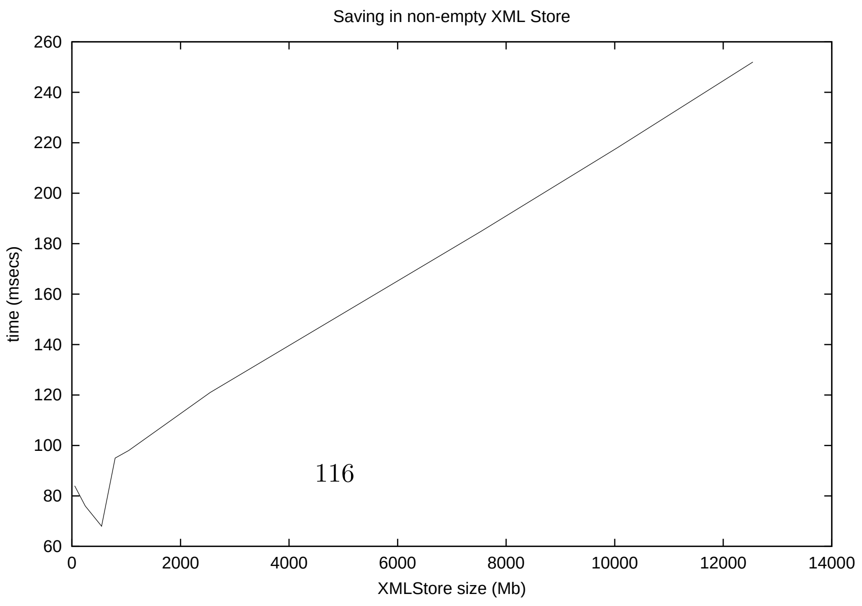
The test results also shows, that the most time consuming operation in the modification is inserting the new XML subtree.

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 (see 6.1.4). The used document have a root with 335 child nodes, and the inserted subtree is added as a child to the root node. The insertion time therefore becomes relatively high. A different strategy for representing child nodes is needed in order to improve the modification performance.


DOM load time, node count fixed at 1000
time (msecs)
XML file size (Mb)
0
200
400
600
800
1000
1200
1400
0
2000
4000
6000
8000
10000
12000
14000


DOM load time, Doc size fixed at 5 Mb
time (msecs)
Number of mappings
1500
1400
1300
1200
1100
1000
900
800
700
600
500
400
0
2000
4000
6000
8000
10000
12000
14000
16000
18000


Saving in empty XML Store
time (msecs)
XML Document size (Mb)
400
350
300
250
200
150
100
0
2000
4000
6000
8000
10000
12000
14000


Saving in non-empty XML Store
time (msecs)
XMLStore size (Mb)
260
240
220
200
180
160
140
120
100
80
60
0
2000
4000
6000
8000
10000
12000
14000

Chapter 9

Future works

Some aspect were beyond the scope of this thesis. These were mentioned in section 1.1.2 as limitations in the development of XML Store. Improvements to existing functionality of XML Store must also be made. Future work will have to address some of these limitations and improvements to provide a usable XML Store.

9.1 Name Service

The evaluation of the name service have shown shortcomings in its functionality. Improvements must be made in three areas:

The limited name space allowed by the service lowers the usability of the name service. Implementing more advanced name service functionality will solve this problem.

The second problem is the missing possibility to implement concurrency control in methods for binding and rebinding. A solution to the problem have been proposed and must be implemented.

The third and last improvement concerns the single point of failure problem in centralized solutions. The name service should be implemented as a decentral distributed service to eliminate the single point of failure problem.

9.2 Stream based API

The loading strategy described in section 6.1.1 is not satisfying for XML document, which either contain XML elements with a huge number of children, or XML character data, which are huge. As mentioned in the section (6.1.1 a solution is to load child nodes and node value lazily, i.e. to introduce a stream based solution.

9.3 Distributed Garbage collection

The current implementation of the XML Store disk layer only allows for saving documents. Deleting documents from disk storage is not possible. As a consequence storage will gradually be filled up, possible with data not used. Introducing some sort of distributed garbage collection solves the problem.

In Distributed Systems Concepts and Design[5] garbage collection of distributed objects is 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 ot it, the object will be collected and the memory it uses recovered

This definition can be transfered to a value-oriented programming model by exchanged 'object' with 'value'. Implementing distributed garbage collection is complicated task, but it has been done [38].

9.4 Network communication

The process of locating documents in XML Store relies on IP Multicast, which is not completely reliable. Messages send on a network might therefore get lost. IP Multicast should therefore be exchanged with a more reliable multicast technique. If multicast techniques are satisfying at all if using XML Store as a 'real' Application Programming Interface is not known, As an example the impact of the number of XML Store peers is not known. Further testing of the multicast solution is thus required.

9.5 Performance issues

The performance tests carried out in chapter 8 revealed that save and load performance could be improved. Section 5.2.4 describes strategies for improving such performance. The ones not implemented are in-lining, asynchronous write and read / write buffering. Implementing these are expected to have considerable influence on load and save performance.

The test of modification performance also revealed low performance, which is due to the naive implementations of child nodes resulting in insertion (i.e. inserting a child into a child nodes representation) times of O(n). Modifying this is simple, but requires a modification of the `ChildNodes` interface

9.6 Querying XML documents

As increasing amounts of information are store, 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 document[16].

Implementing a query language for XML Store is provides 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. XQuery could be implemented for querying documents in XML Store.

9.7 Mobility

The world is 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 be supported on such computing devices frequently moving from one physical location to another, it must be able to support that devices, which documents are stored on, change physical location.

XML Store and its Document Value Model interface has taken advantage of the value oriented model by providing an interface in which document location is transparent. (Document are loaded and saved using value references). The Document Value Model thus supports an extension of the XML Store functionality to support mobility of devices (being XML Store peers) persisting XML data.

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. Besides disk and network initializations the implementation of the Document Value Model (DVM) must be chosen.

XML Store *properties* are values used for initializing an XML Store. The properties are stated in a *property file*. 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, that is the Java class `java.util.Properties`.

- 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

# SERVER INFORMATIONS:
#name server ip
Name_server_IP = 228.5.6.10

# 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
```

The default property file is always to be named "xmlstore.properties" and to be situated in the "etc" folder next to the xmlstore.jar file. Different property files can be created. They must have the extension ".properties".

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 (as a Document Value Model) and an application, `XMLStoreDictionary`, using this XML document have been developed. `Node-Counter` is a simple application counting nodes in XML documents, which are store in XML Store. The following pages contain source code for the sample applications.

Appendix C

Source code

The source code for the XML Store implementation is presented on following pages.