A distributed, value-oriented XML Store

Master's Thesis
IT University of Copenhagen

Tine Thorn
Anders Baumann
Mikkel Fennestad

Supervisor:
Peter Sestoft

August 2002

Abstract

The XML Store is a distributed, value-oriented storage facility for storing XML documents. This thesis demonstrates that it is possible to design and implement an XML Store based on a peer-to-peer distributed file system. The distributed file system is based on Chord, a routing and location protocol developed at MIT, and provides a distributed hash table for storage of data. The Chord protocol is highly scalable as it looks up data in time logarithmic in the number of servers and requires only logarithmic space for routing information at each node.

The term value-oriented denotes a style of programming where all data are considered immutable. When storing XML documents we employ a storage strategy, splitting XML documents up according to their inherent tree structure. Each subdocument is considered a value and stored separately in the distributed file system. This storage strategy offers many advantages over conventional text based technologies such as SAX and DOM, the most important being the possibility to obtain sharing of identical subdocuments.

We present a simple and flexible value-oriented API for representing and manipulating XML documents. It takes advantage of sharing and uses the proposed storage strategy to handle manipulation of XML documents without having to read and parse the entire document into memory. The value-oriented approach makes it easier to maintain several document versions and simplifies usually complex problems in distributed systems, such as transaction handling, caching- and replication management.

The implemented prototype has been subjected to extensive performance experiments. The experiments demonstrate the superiority of the value-oriented approach when storing modifications of documents. The experiments show that the time taken to store a document is mostly dependant on the number of nodes in a document, and to a lesser degree the size of the nodes. The experiments furthermore reveal that the XML Store prototype does not perform well when storing nodes. This is probably due to the somewhat rough and unoptimised implementation of network communication. With a better implementation it is plausible that the XML Store can be much more efficient, since related systems based on the Chord protocol achieve performance comparable to FTP.

Contents

Preface

This Master's Thesis has been written in the period from February to August 2002 at the IT University of Copenhagen under supervision of associate professor Peter Sestoft.

The thesis concerns the design and implementation of a distributed, value-oriented storage facility for XML documents. This work relates to existing results on peer-to-peer storage facilities. However, the idea of implementing a distributed storage facility specifically designed for storing XML documents in a value-oriented way is novel.

The thesis has been written in British English. Text that refers to code or code related items are written in `verbatim` font. References are cited by index in the bibliography, e.g. [1]. If a reference appear after a full stop, it means that this reference applies to the preceding paragraph.

The appendix referred to throughout the thesis can be found in a separate document. Source code of implementations can be found in the appendix. Source code is also available for download at the web site:
`http://www.it-c.dk/people/fenne/xmlstore/src.zip`

Thesis handed in August 1st, 2002.

Acknowledgements

A number of people directly or indirectly influenced this work and we are grateful for their contributions.

We would like to thank our supervisor Peter Sestoft for skillful and patient guidance, support and advice during the course of writing this thesis. We would also like to thank associate professor Fritz Henglein for providing the initial impetus inspiration for "Plan 10", and the "XML Store".

Furthermore, we acknowledge Jesper T. Pedersen and Kasper B. Pedersen for fruitful discussions in the area of value-oriented programming and XML storage. A special thanks to Jakob Bendsen and Rasmus Lund for insight, diversion and support, and to Jacob Fennestad for design and creation of the frontpage.

Chapter 1

Introduction

This chapter motivates our interest in distributed (peer-to-peer) systems for storing XML documents in a value-oriented way. Furthermore, we present the main purpose of the thesis, some desiderata of the XML Store system and delimitations of the thesis. Finally, we give an overview of the report.

1.1 Motivation

Distributed Systems

Not long ago computers were a scarce resource, but in recent years the cost of processors, memory, storage space and fast network connections have fallen dramatically. Networked computer systems are rapidly growing in importance as the medium of choice for the storage and exchange of information and data.

The growth in storage, bandwidth, and computational resources has fundamentally changed the way that applications are constructed, and has inspired a whole new class of distributed, peer-to-peer storage infrastructures. Peer-to-peer storage systems such as CFS [1, 2] and PAST [3, 4] seek to take advantage of the rapid growth of resources to provide inexpensive, highly available storage without centralised servers, as centralised servers are considered a single point-of-failure and subject to crashes, denial of service attacks and unavailability due to regional network outages [5].

There are of course drawbacks to the peer-to-peer architecture. One is the lack of control over the network. Security policies, backup policies and so on are complicated to implement in a peer-to-peer network, and you run the risk of having malicious peers join the system and try to subvert the system. Another disadvantage is that peers tend to be more unstable than servers and will fail more frequently.

Overcoming these challenges will however make it possible to build interesting and useful applications that harness the power of the many computers connected to the Internet.

XML

The amount of data of all kinds available electronically has increased dramatically in recent years, with the expansion of the Internet and the increased interconnectedness between computers. This has introduced problems with the way data are stored and exchanged.

First of all, formats differ to a great extent. Data reside in countless different forms, ranging from unstructured data in file systems to highly structured data in relational database systems. Furthermore, applications often store data in proprietary non-standard formats. Secondly, data often stem from heterogenous data sources, belonging to external organisations or partners, not under the application's control. Sometimes the structure is only partially known, and may change without notice. [6, 7]

Research on XML started with the observation that much of today's electronic data do not conform to traditional relational or object oriented data models. An XML document is an instance of semistructured data, which are data that cannot always be constrained by a schema. XML is basically a linear syntax for expressing tree structured data and has become widespread to the point of being the de facto-standard for data exchange over the Internet.

Value orientation

The term "value-oriented programming" specifies a way of working with data and variables that differs from the prevalent approach known from imperative programming languages. The designation "value" describes an entity that cannot be altered. Value-oriented programming is known from functional programming languages such as SCHEME, Standard ML and Haskell that origin in the lambda calculus [8].

The central idea of value-oriented programming is that data are immutable, and you manipulate data by creating new values, not by altering existing data. In traditional programming languages data are instead modified destructively and consequently the original values are lost.

Most distributed systems have adopted the way of working with data from the way imperative programming languages handle values – i.e. destructive updates of data. However, this induces problems in a distributed environment. Caching and replication of data are needed in distributed systems to provide fast access to shared data (caching) and increased fault tolerance (replication). But these techniques introduce problems with inconsistency if data can be updated. The system must ensure that the concurrent execution of actions on replicated data is equivalent to a correct execution on non-replicated data. This leads to complexity and performance problems when multiple clients modify shared data.

In a distributed setting value-oriented programming has an *a priori* advantage over imperative programming, as it does not suffer the problems of handling multiple imperative updates. Cache management and consistency protocols are unnecessary in a value-oriented environment as data are immutable, and updates can be performed atomically in a distributed environment without complex transaction management.

Distributed value-oriented XML Store

The proposed XML Store combines the above technologies. It uses an efficient, distributed peer-to-peer file system for permanent storage and offers a rich value-oriented API for traversing and manipulating XML documents.

Even though XML is quite widespread today, the conventional ways of working with it seem somewhat undeveloped. XML documents are usually stored in flat text files and manipulated using technologies such as SAX and DOM. By storing documents in flat text files you only have serial access to XML documents and this approach requires repeated serialisation and deserialisation from object to text representation: Data are copied from disk and parsed to an abstract representation that the programmer can work with. After the document has been processed and possibly modified, it is again transformed to a textual representation and serialised to disk. This is obviously inefficient and inflexible.

The XML Store offers a more flexible and sophisticated way of working with XML. By working value-oriented and storing the documents according to their inherent tree structure a lot of possibilities arise for more elegant XML processing, for example: The document can be traversed on disk thereby removing the need for loading the entire document into RAM. Parts of a document can be loaded only when required – so-called *lazy loading*. By splitting the document up into subdocuments identical parts can be shared between different documents, in RAM as well as on disk. And when moving to a distributed setting caching and replication is easily achieved due to the lack of need for a coherence protocol.

1.2 Problem statement

The main purpose of this thesis is to demonstrate that it is possible to design and implement a value-oriented XML Store based on a peer-to-peer distributed file system. The distributed file system is based on Chord, a distributed routing and location protocol, which allows storage and retrieval of data. We propose a storage strategy that divides an XML document in subdocuments according to the XML document's inherent tree structure and consider each subdocument an immutable value. The value-oriented storage strategy offers many advantages over existing technologies for handling XML, the most important being the possibility to obtain sharing of identical subdocuments. We propose a flexible, value-oriented API that takes advantage of sharing and uses the proposed storage strategy to handle manipulation of large XML documents without having to read and parse the entire document into memory.

The implementation of the XML Store will serve as a proof-of-concept prototype, that can be used for tests and evaluation, though it may not achieve the performance and fault tolerance of complete systems. We focus on obtaining the proper asymptotic complexity in its essential operations. Although we do not focus on constant factor optimisations, the system must have an acceptable performance in a realistic environment.

By designing and implementing a value-oriented API for storing, retrieving and working with XML documents on top of a peer-to-peer file system based on the Chord protocol, we intend to:

1. Put the concept of a value-oriented XML Store into practice, thereby either verifying its potential or recognising flaws in the concept.

2. Present a rich and flexible API for value-oriented XML processing. To evaluate the applicability and adequacy of the API, we build a distributed e-mail system to clarify whether the API is adequate or lacks features that an application programmer would consider essential.

3. Determine how well suited the Chord protocol is for building a distributed, value-oriented storage facility and evaluate the efficiency of the proposed storage strategy. We do this by carrying out a number of experiments involving storage and retrieval of different types of documents, and analyse the consequences of working value-oriented and using sharing.

1.3 Desiderata

In this section, we list a number of *desiderata* (desired properties) of the XML Store system. It is obvious that some of the desiderata are mutually exclusive – for instance, it may not be possible to obtain both high performance and a complete decentralised system and it is probably difficult to provide a global search facility while ensuring that only authorised users have access to data. The order of desiderata does not indicate priority.

- ***No single point of failure*** The XML Store system should be fully distributed and there should be no single point-of-failure in the system.

- ***Scale gracefully*** The XML Store system should scale gracefully when the number of peers and amount of data increases.

- ***High performance*** The XML Store system should perform well, when saving and loading data (even under heavy load).

- ***Fault tolerant*** The XML Store system should be able to recover from failures, if such occur.

- ***Available*** The XML Store system should guarantee availability of data stored in it.

- ***Load balance*** The XML Store system should balance load evenly among the peers in the system.
- ***Self-organising*** The XML Store system should be self-organising, meaning that it should be able to automatically adapt to the arrival, departure and failure of peers.
- ***Secure*** The XML Store system should ensure that only authorised users get access to data and resources found in the system, thereby protecting against the following three security threats:
  - *Leakage*: Acquisition of data by unauthorised users.
  - *Tampering*: Unauthorised alteration of data.
  - *Vandalism*: Interference with the proper operation of a system (e.g. by denial of service attacks).
- ***Simple*** The XML Store system should be simple to implement, meaning that a simple design, simple algorithms etc. are desirable.
- ***Efficient XML processing*** The XML Store system should provide features for manipulating XML documents efficiently.
- ***Applicable API*** The API of the XML Store should be simple, easy to use and should provide features (methods) that an application programmer will find adequate.
- ***Search facility*** The XML Store should support searching of the stored documents. Search is a useful operation when wishing to find data without prior knowledge of its exact name or location. The XML Store should facilitate different types of searches, ranging from finding a document with a certain name or certain content to structured database style “queries” in XML documents.

1.4 Contributions

Henglein [9] provided the initial idea of a value-oriented “XML Store”, which is part of the ongoing “Plan 10” project. The XML Store is a distributed infrastructure for value-oriented storage of XML documents.

We have implemented a fully working prototype of the XML Store and combined the idea with a distributed peer-to-peer system based on the Chord protocol. Implementing a distributed storage facility specifically designed for storing XML documents in a value-oriented way is novel. By implementing the XML Store prototype, we have verified that the fundamental idea of an XML Store is realisable and has many promising aspects.

By evaluating the XML Store prototype we have shown that there are substantial advantages to be gained from storing XML documents in the XML Store compared to current technologies, such as SAX and DOM. Furthermore, we have identified a number of areas that need to be improved for the prototype to grow into a fully operational system.

We have designed and implemented an API for working efficiently with XML documents in a value-oriented way, and have demonstrated how to use the API by implementing a distributed e-mail application.

Furthermore, we have illustrated that the value-oriented programming model in general has advantages in distributed systems.

1.5 Delimitations

Designing a distributed XML Store that fulfills the desiderata mentioned earlier is an impossible task, since some of the desiderata are mutually exclusive. We concentrate on designing a prototype to demonstrate important principles and gain insight, instead of focusing on completing and optimising a small number of modules.

Throughout the thesis we will point out topics that will not be considered further or which are only discussed but not implemented. In the following we will summarise the delimitations of this thesis.

We do not treat the problem of distributed garbage collection and we do not discuss facilities for searching and querying XML documents.

A range of topics are described, but have not been implemented in the prototype. These topics include fault tolerance, security, network locality, optimising load balancing by means of virtual servers and collision detection protocols when using cryptographic hashing. Except for collision detection all of these features have been implemented and described by Dabek in the CFS system [1, 2], and not much is gained by repeating the implementation here. Their ability to function in the context of the XML Store framework is analysed, though.

Some modules of the prototype are clearly not optimally implemented – only highly vulgarised implementations are provided. This includes the implementation of network communication, local disk handling and name service. We analyse the problems with the design of the current modules and sketch better solutions, but will not go into a detailed design.

1.6 Thesis overview

This thesis is organised in three parts: *Background*, *Analysis & design* and *Implementation & evaluation*.

Background consists of four chapters providing background knowledge of topics that we find important for the understanding of this thesis. The chapters are: Distributed systems (chapter 2), XML (chapter 4) and value-oriented programming (chapter 3). If one is familiar with these topics, they can be skipped. Furthermore, this part of the thesis presents a survey of work closely related to ours (chapter 5).

Chapters 6 through 13 constitutes the *Analysis & design* part of the thesis. More specific, we give an overview of the XML Store system in chapter 6. Chapter 7 provides a description of our API. The routing and location scheme, Chord, is discussed in detail in chapter 8. In chapter 9 we present storage strategies and discuss disk handling and chapter 10 provides a discussion of symbolic names and a presentation of our name service. In chapter 11 and 12 we discuss network communication strategies and security issues, respectively. Finally, chapter 13 provides a summary of the second part of the thesis and outlines properties of the XML Store system.

Implementation & evaluation begins with a description of the program and a brief discussion of test strategies (chapter 15). We continue by presenting the experimental results obtained with the prototype implementation of the XML Store (chapter 16) and an evaluation of our API (chapter 17). Finally, we present conclusions and potential future work in chapter 18 and 19.

Part I

Background

Chapter 2

Distributed systems

This chapter defines and discusses some central characteristics of distributed systems. Some key problems that have to be taken into consideration when designing a distributed system, are also discussed. Finally, a special kind of distributed architecture, namely the peer-to-peer architecture, is examined, since this is the architecture that we base the XML Store on.

2.1 Characteristics of distributed systems

The motivation for constructing and using distributed systems stems from a desire to share resources. There is yet no agreed definition of a distributed system [10], but according to Coulouris et al. [5] a distributed system can be defined as follows:

Definition *"We define a distributed system as one in which hardware or software components located at network computers communicate and coordinate their actions only by passing messages".*

The above definition outlines the difference between a distributed system and a local (monolithic) system. In a monolithic system hardware or software components do not communicate via a network. The term *messages* covers both messages containing simple data such as state and messages containing more complex data, such as programs.

Since computers in a distributed system may be spatially separated by any distance, the definition of a distributed system has some significant consequences: Concurrent activity, lack of global clock and independent failures [5]. In the following we briefly describe the consequences.

Concurrency

Typically in a distributed system, computers – or rather processes on computers – share resources, such as files. It is therefore important that a distributed system can handle several processes concurrently accessing a shared resource without blocking each other or corrupting the shared resource [5].

Lack of global clock

In contrast to processes that run on one computer, processes that run on several computers in a distributed system do not have a common global clock [5]. Computers in a distributed system cannot completely synchronise their clocks since they are communicating and coordinating their actions by passing messages. Since messages are sent between computers in a network and since it is typically not possible to determine exactly for how long a message has been on its way, it is difficult to synchronise clocks on different computers (even though there are protocols that seek to assure an accurate synchronisation, e.g. NTP [11]). This means that in practice, communication between computers in a network is asynchronous.

Independent failures

In a distributed system each component of the system can fail independently of the rest of the system [5]. This means that it is possible for a single component (e.g. a computer process or network connection) to fail while the remaining components continue to function. When an independent failure occurs it may not be possible for the remainder of the system to detect whether the network has failed or has just become very slow due to heavy load. Failures of a computer is not immediately made known to the other components of the system with which it communicates.

2.2 Challenges designing distributed systems

When designing a distributed system there are some key challenges one has to deal with or at least should be aware of. In the following we will discuss some of these challenges and their solutions as stated in Coulouris et al. [5].

Heterogeneity

In distributed systems it is often necessary that heterogenous software and hardware work together. Programs that are part of the distributed system may be written in various programming languages, they may run on different kinds of computers having different operating systems, capacities etc. and different types of network may be involved. Furthermore, very often different developers and organisations contribute to the implementation of a distributed system.

To accommodate a heterogenous environment in a distributed system middleware layers may be introduced so that the heterogeneity is masked.

Scalability

Often many clients need to share the same resource in a distributed system. Scalability concerns the ability of a system to accommodate an increasing number of concurrent clients that access some resource or work together.

A system is described as scalable if it remains effective when there is a significant increase in the number of resources and users [5]. An effective distributed system should be able to easily accommodate expansions as well as reductions [12].

Concurrency management

As mentioned earlier, a distributed system consists of processes, placed on different computers, that may share some resources. If several processes concurrently modify the same resource it may become inconsistent, and therefore concurrency must be controlled in a distributed system.

There are different kinds of concurrency control techniques, e.g. *locking* and *timestamp ordering* [13].

A typical solution to the concurrency problem is to use transactions, which are atomic units of processing that are either completed entirely (*committed*) or not at all (*aborted* or *rolled back*) [13]. A transaction may include one or more operations on a shared resource (i.e. insertion, deletion, modification and retrieval operations) [13]. Transactions submitted by various users may execute concurrently, and therefore concurrency control is needed to ensure that concurrent transactions will execute in a controlled manner.

However, if the resource is distributed due to replication or caching, concurrency control becomes considerably more complex, since locking of a resource is no longer sufficient for preventing inconsistencies.

Failure handling

Handling failures in a distributed system can be quite difficult, because it can be hard or even impossible to detect failures. If a failure has been detected, it is necessary to decide on a proper response. Such a response could be to retry the failed event (i.e. recovery). If a server crashes in the middle of processing data, the state of data need to be recovered or “rolled back”, to make sure that data will remain in a consistent state.

Typically it is infeasible to try to detect and correct every failure that might occur in a large distributed system. Therefore, it may be better if some failures are tolerated. One way to make systems fault tolerant is by using redundancy.

Openness

It is often necessary to change a distributed system, e.g. when parts of a distributed system have to be replaced by a new implementation which offers better support of security, failure handling and so on. Also when new types of clients need access to a shared resource, it may be necessary to change the distributed system. Coulouris et al. [5] mention *open standards* as the best way of supporting openness. Open standards means standards which are documented and public available (they are independent from individual vendors), and thereby possible to implement on various systems. Some examples of open standards regarding distributed systems are TCP/IP and UDP/IP. These open protocols make it possible to connect new systems to existing systems as long as they support the protocols, regardless of the actual network and operating system.

Security

When data are stored in a distributed system security must be taken into consideration because of the importance and intrinsic value that the data might have to the owners/users of the data. Some of the main challenges concerning security that Coulouris et al. [5] mention are:

- Confidentiality: Confidential data should only be available to authorised users (protection against leakage).
- Integrity: It is important that vicious individuals/users cannot destroy, alter or corrupt data (protection against tampering).

Both of the above two challenges can be met by the use of encryption. With encryption techniques it is possible to conceal information in a message and to authenticate and identify another user somewhere in the network.

Another challenge concerning security is protection against vandalism, such as denial of service attacks. It is important to be able to prevent such *virus-style* attacks on a server.

Transparency

Coulouris et al. [5] define transparency as “The concealment from the user and the application programmer of the separation of components in a distributed system so that the system is perceived as a whole rather than as a collection of independent components”.

As can be deduced from the above definition, the purpose of transparency is to make it easier for users and application programmers to use and work with a distributed system – they will not get confused.

As an example of transparency, consider Java RMI, where method invocations on remote objects are very much like local method invocations, as they use the same syntax.

2.3 Peer-to-peer systems

In recent years peer-to-peer systems have become quite popular as they offer some advantages over traditional client-server systems, albeit often at different fields of application. In this section some of these advantages are described and some of the challenges that the peer-to-peer architecture introduces are outlined.

The designation *peer-to-peer* refers to the topology and architecture of the computers in a distributed system [14]. Peer-to-peer systems can be characterised as decentralised distributed systems in which all participants have equal status, and where all communication is symmetric [3, 15, 16].

Decentralisation of a system prevents a small group of participants from limiting the overall performance of the system. "Equal status" and "symmetric communication" means that in contrast to the client-server model, there is no distinction between servers and clients in the peer-to-peer model: A peer can both request a service and provide a service [16]. Figure 2.1 illustrates a peer-to-peer network, consisting of different kinds of peers.

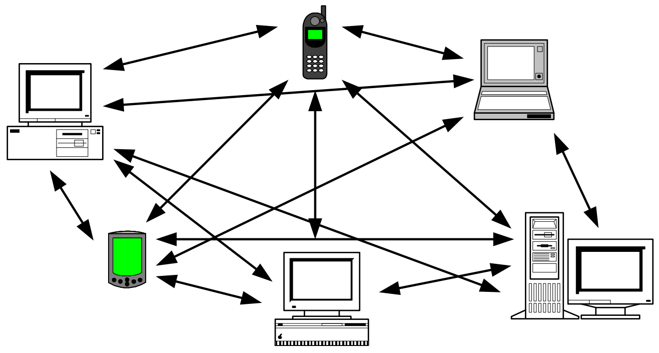


Figure 2.1: Illustration of a peer-to-peer network.

Existing peer-to-peer systems, such as Freenet [17] and Gnutella [18], demonstrate the benefits of cooperative storage and serving. One of the primary advantages of decentralised peer-to-peer systems over client-server systems, is their extensibility [16, 19].

Figure 2.2 illustrates a typical client-server architecture. As one might can imagine, the pressure on the central server increases when clients are added to the client-server network. When adding more peers to a peer-to-peer network the capabilities of the system grows stronger, because there is no single point-of-failure and each peer contributes resources, such as storage space and CPU cycles, to the overall system [16]. Peer-to-peer systems have the ability to utilise idle storage and network resources of large numbers of participating computers [1]. They also have the potential of being more fault tolerant than their client-server counterparts, as they do not have a single point-of-failure [2, 19]. Furthermore, peer-to-peer

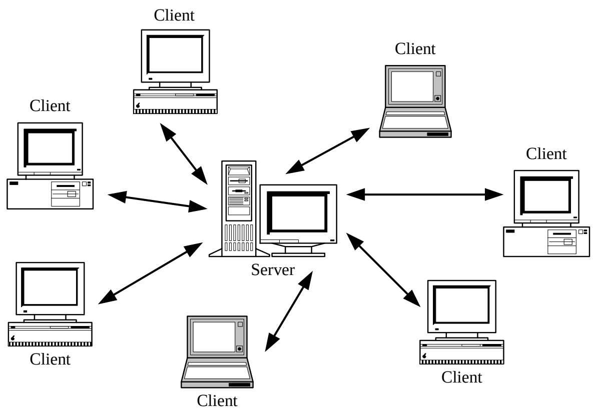


Figure 2.2: Illustration of a client-server network.

systems tend to have a higher degree of availability, since a large number of peers can crash or leave the system, without destroying the whole system. In theory only *one* peer is needed for a peer-to-peer system to be "alive" [16]. In general, it is quite difficult to evaluate the scalability of decentralised systems such as peer-to-peer systems. Theoretically, the more peers you add, the more capable a peer-to-peer system should be. However, in practice it is necessary to keep the system coherent, and algorithms responsible for this often carry a lot of overhead [19]. Therefore, peer-to-peer systems might not scale well.

Challenges and drawbacks

There are also some challenges and drawbacks associated with the design of peer-to-peer systems. Peers in a peer-to-peer network are less enduring than servers in a client/server network, and will join and exit the network at will. This entails that the underlying peer-to-peer protocol must be able to adapt to frequent changes in the networks configuration without affecting robustness and efficiency of the system [1].

One drawback to the peer-to-peer architecture is the lack of centralised control over the network. As a consequence peer-to-peer systems tend to be hard to manage [19]. In a client-server network administration is only needed at the central points. With the peer-to-peer architecture a number of organisational, administrative etc. problems emerges. In this thesis, however, we will focus on the technical problems related to the peer-to-peer architecture.

Peer-to-peer systems also raise some interesting security problems. Making your computer available to any other peer-to-peer participant and allowing them to upload to and download from your computer requires a level of trust that most people find uncomfortable [20]. Since security policies can be hard to implement in peer-to-peer systems, these systems tend to be insecure [19].

However, according to both Dabek et al. [21] and Rowstron & Druschel [22] the core technical problem facing large-scale peer-to-peer systems is to provide efficient means for data locating and routing within the decentralised network.

We have now presented peer-to-peer systems and distributed systems in general. A discussion and survey of specific peer-to-peer systems is presented in chapter 5.

Chapter 3

Value-oriented programming

This chapter describes the basic concepts of value-oriented programming, which is the programming model used by the XML Store.

Value-oriented programming is programming with values and references to values (value references), which are values themselves. Values are per definition immutable, e.g. the value 5 will always be 5. This principle does not only apply to primitives but also to complex and composite values such as lists and tree structures. Hence, it is impossible to change any constituent of a composite value.

Programming with immutable values may, for the purely imperative programmer, seem alien and not very convenient. As the imperative paradigm revolves around assignment and hence modification of data, not being able to update may seem like a limitation in the programming model. This is however not the case.

The imperative programming model has a “copy-and-update” style of data manipulation, which typically involves the following three steps:

1. Copy data from source.
2. Destructively update the copy.
3. Copy the updated data to source.

Value-oriented programming adopts a “share-and-create” style known from functional programming. It is characterised by sharing as much data as possible and only copy data if there is a need for update [9]. The three steps from above correspond to:

1. Take a reference to data.
2. Nondestructively create a new value, which typically involves copying parts of the original data.
3. Replace the original reference with a reference to the new value.

It is not necessary to move to a functional programming language, to find an example of the value-oriented concept. Java’s String API is value-oriented. Strings in Java are treated as immutable objects, which means that all modification methods create a new String object, leaving the original object unchanged. An example which illustrates the "create" part of the "create-and-share" style is the concatenation of two String objects. In Java this is accomplished by creating a third String object, leaving both original strings unchanged. The "share" part refers to the use of references to existing data, and is possible because of value references and the immutable nature of values. Value references and sharing are explained in the following section.

3.1 Value references and sharing

A value reference is a unique identifer for a value. According to Henglein [9] it has the following desired properties:

- A value reference is a deterministic function of the value alone.
- A value reference is injective, meaning that distinct values are mapped to distinct value references. As a consequence of this, a value reference is as immutable as the value itself – it cannot be updated to refer to a different value.
- Whereas a value can be arbitrarily large, a value reference has a limited size.

Since a value reference is independent of the location of the value, a reference resolver is necessary to retrieve the actual value. The Chord protocol is an example of a reference resolver in a distributed environment (Chord is explained in detail in chapter 8).

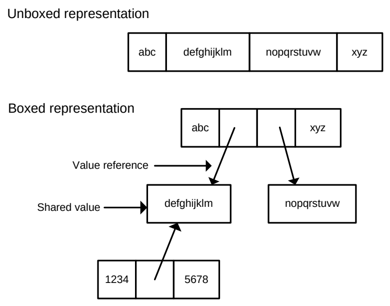


Figure 3.1: A boxed and unboxed representation of a list of values. The figure also shows sharing of values.

Consider the unboxed representation of a list in figure 3.1. Because of the deterministic relationship between a value and its value reference, they can always be substituted for each other. This is illustrated in the boxed representation in figure 3.1, where two values have been replaced by their value references. The technique of boxing a value is also called *value coalescing*: copies of data are replaced by references to a single instance of the data [9].

Value references (boxes) allow efficient sharing of values. Figure 3.1 shows an example of sharing, where a value is shared between two lists. Observe that sharing is only possible because value-oriented programming provides a semantic guarantee: *No updates*.

3.2 Value-oriented trees

This section describes how to work value-oriented with tree-structured data. We focus on tree-structured data, because an XML document is a linear syntax for describing labeled trees (see chapter 4.1 for a detailed description of XML).

Consider the two trees in figure 3.2. Instead of having two isomorphic subtrees, one of the subtrees is replaced by a reference to the subtree. In this way the two trees form a directed acyclic graph (DAG), as a node can now have more than one parent. It is still possible to recreate the original two trees by traversing the DAG from the two nodes that are roots in the trees.

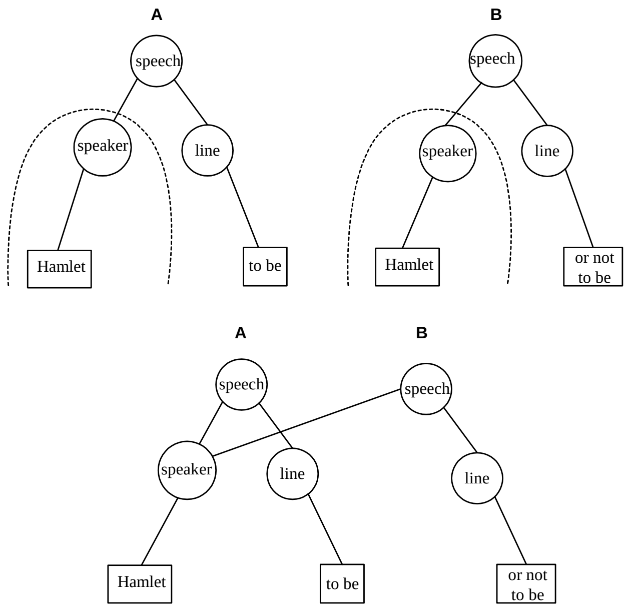


Figure 3.2: Isomorphic subtree shared by two trees A and B.

In value-oriented programming every single node in a tree can be considered an immutable value. Modification of a tree is accomplished by creating a new and modified tree and sharing unchanged parts. Figure 3.3 shows tree representations of two XML documents, A and B. In document A we want to change the content of node $Hamlet$ to $Ophelia$. A modification of a node propagates all the way to the top of the tree, which means that a new tree A' has to be built from the root of A. This is less wasteful than it sounds since A' shares most of its nodes with A and at most $depth(A)$ new nodes are introduced in A'.

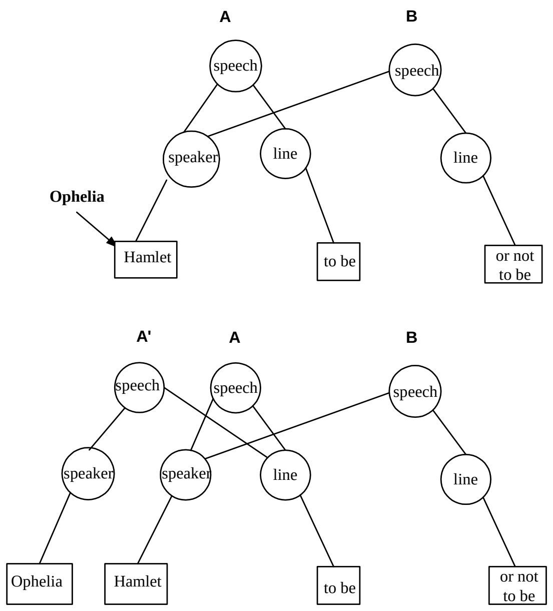


Figure 3.3: Modification of a tree using value-oriented programming.

The rest of the basic operations for tree manipulation are accomplished in a similar way as update: To delete a subtree a new tree is built without the subtree that we want to delete, and to add a node a new tree is built with the new node added to the desired subtree.

Chapter 4

XML

Since our data storage is based on XML we briefly describe the XML standard in this chapter. We also give brief, informal introductions to a number of XML related technologies that are relevant for this work, namely a simplified form of XML called Minimal XML, and SAX, DOM and XPath. If the reader is already familiar with these technologies, the chapter can be skipped.

4.1 What is XML?

XML (*Extensible Markup Language*) is a standard for document markup, defined by the World Wide Web Consortium (W3C) in February 1998 [23]. It defines a standard syntax for markup of data with simple tags.

XML can be considered an instance of *semistructured data*. Semistructured data are often referred to as "self-describing" or "schema-less", which means that neither the structure of data nor the type of data is required to be described separately [6, 24, 25]. Instead semistructured data are directly described using a simple syntax (XML uses tags for instance), and thus there is no need for an *a priori* schema as in relational or object-oriented data models. [25] Semistructured data may have an incomplete or irregular organisation indicating that semistructured data are more suitable for modelling heterogeneity than both the relational and object-oriented model [25, 26].

Data exchange and publication on the Web often involves heterogenous data, which is one of the reasons why XML has become widely distributed on the Web. XML has in recent years become a de facto standard for data representation on the Web, providing a simple data syntax which is both human- and machine-readable [25].

XML is an instance of SGML (Standard Generalised Markup Language) [23], just as, the arguably most widespread markup language, HTML (HyperText Markup Language) is. The main difference between HTML and XML is that in HTMl there is a fixed set of tags and semantics. XML on the other hand is a meta-markup language [27], which means that it specifies neither semantics nor a tag set. XML is therefore completely extensible, and more regular than HTML, as it allows users to define their own *elements*. Elements are the basic building blocks of XML. They may contain text, comments, and other elements, and consist of a start tag and an end tag.

Figure 4.1 shows an example of a simple XML document. The document has four different tags, namely `play`, `title`, `personae` and `persona`. The `title` element contains simple text, whereas the `personae` element has two `persona` elements both containing text. The root element of the document is `play`. The `title` and `personae` elements are considered children of `play`, and the two `persona` elements are considered children of `personae`.

```
<play>
    <title>
        The Tragedy of Hamlet, Prince of Denmark
    </title>
    <personae>
        <persona>
            HAMLET
        </persona>
        <persona>
            OPHELIA
        </persona>
    </personae>
</play>
```

Figure 4.1: The figure shows a simple XML document containing five elements, the root element being `play`.

4.1.1 Well-formed XML

XML documents must be *well-formed* to be parsed correctly [27]. A well-formed XML document is one that has exactly one root element, has every tag closed (i.e. every start-tag must have a matching end-tag), and where no overlapping between tags occur (no interleaving). It is perfectly alright to embed an element in another element, but then this element must be completely enclosed by the other element. That is, both start tag and end tag must be inside the other element.

Order

The contents of an XML document are by definition ordered. This means that the order of the children is crucial to the sense of the XML document. Since semistructured data are considered unordered, XML and semistructured data are thus different with respect to this characteristic [28].

Tree-representation of XML

A well-formed XML document represents a tree structure. Embedded elements reflect a parent-child relationship, where the embedded elements can be viewed as children of the enclosing element. The single topmost element of an XML document has no parent. This element is the root element and it contains all other elements in the document.

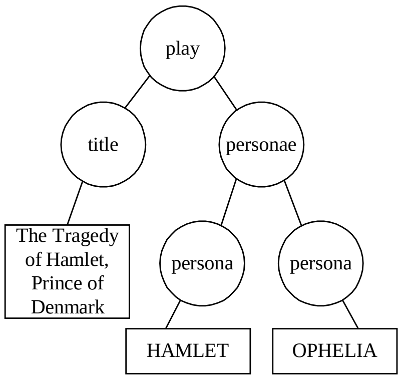


Figure 4.2: Tree representation of the XML document found in figure 4.1.

The tree in figure 4.2 represents the XML document presented in figure 4.1. The root element of the XML document is the `play`-element, and the root node of the XML tree is consequently a node labeled *play*.

4.1.2 Valid XML

An XML document can optionally conform to a Document Type Definition (DTD), that establishes a set of constraints for an XML document. An XML document is considered *valid* if it follows the constraints that the DTD imposes. [27]

Figure 4.3 shows a DTD for `play` documents (shown in figure 4.1), which states that a `play` element contains one `title` child element and one `personae` child element. It also states that both `title` and `persona` elements contain text, and that `personae` elements contains an arbitrary number of `persona` elements. The expression `persona*` is a regular expression, meaning any number of `persona` elements. Other common regular expressions used by DTD are e^+ (one or more occurrences of e), $e?$ (zero or one occurrence of e), $e1 \mid e2$ ($e1$ OR $e2$, alternation) and $e1, e2$ (both $e1$ AND $e2$, concatenation).

4.2 Minimal XML

"One should not increase, beyond what is necessary, the number of entities required to explain anything." William of Ockham (1285-1349)

```
<!ELEMENT play      (title, personae)>
<!ELEMENT title     (#PCDATA)>
<!ELEMENT personae  (persona*)>
<!ELEMENT persona   (#PCDATA)>
```

Figure 4.3: Example of a DTD for the `play` document.

In this section we introduce Minimal XML, since we use this markup language in the XML store for the sake of simplicity. Minimal XML[1] is a simplified version of the XML language [29, 30]. According to Park [31], the purpose of making an even simpler form of SGML than XML was to make:

- A subset that allows easily implemented parsers that are much faster and smaller than full XML parsers.
- A subset with a simpler information model that can easily be mapped to other information models.
- A subset that is much easier to learn, teach and use.

Several XML features are omitted from Minimal XML. Some of these are: Attributes, Processing Instructions, Document Type Declaration, Non-character entity-references and CDATA marked sections. The most significant difference when modelling data in Minimal XML is the lack of attributes to hold information. Instead child elements can be used to hold information (see figure 4.4). The first part of the figure (A) shows an XML document which uses attributes to hold information about the speaker of the line. This document does not conform to the Minimal XML standard since an attribute is used. The second part of the figure (B) illustrates an XML document which uses a child element (*speaker*) to hold the same information. This document conforms to the Minimal XML standard.

4.3 XML technologies

The following presents the XML technologies SAX, DOM and XPath. SAX and DOM are arguably the most predominant technologies for working with XML documents, and the advantages and drawbacks associated with them will therefore be discussed. XPath is a specification for addressing specific parts of an XML document using path based regular expressions. In the value-oriented API we use XPath and we therefore briefly describe the syntax.

[1]Minimal XML was previously known as Simple Markup Language (SML).

```
A  <scene>
      <speech speaker="HAMLET">
         <line>
            To be, or not to be: that is the question:
         </line>
      </scene>
   </scene>

B  <scene>
      <speech>
         <speaker>
            HAMLET
         </speaker>
         <line>
            To be, or not to be: that is the question:
         </line>
      </speech>
   </scene>
```

Figure 4.4: Part A does not conform to the Minimal XML standard while part B does.

4.3.1 SAX and DOM

SAX

The *Simple API for XML* (SAX) is an example of an event based framework for parsing XML documents. SAX is not itself an XML parser, but a defines a set of callback methods that are to be called by the parser when an event occurs, such as when the parser encounters the start tag and end tag of elements. To use SAX, an application programmer registers a SAX handler with the parser and begin parsing the document. The handler is notified of events when the callback methods are invoked by the parser. [23, 32]

An XML document is thus presented as a linear sequence of events. Table 4.1 shows the events generated by the SAX parser when parsing an XML document. SAX provides no abstract representation of the document, only events, which makes it difficult to work with and modify an XML document.

DOM

The *Document Object Model* (DOM) is an example of an API providing an abstraction of XML documents. In DOM, XML documents are represented as objects in a tree structure. DOM defines interfaces for each different entity in an XML document (elements, character data blocks, attributes, etc.), and specifies methods for traversing the structure and manipulating the document.

Unlike SAX, that provides stream-based access to the XML data, DOM provides tree-structured access to nodes in the object representation. Furthermore, DOM provides ways to manipulate XML data, whereas SAX only provides access

Document	Event
`<play>`	begin ”play”
`  <title>`	begin ”title”
`    The Tragedy of...`	character data ”The Tragedy of...”
`  </title>`	end ”title”
`  <personae>`	begin ”personae”
`    <persona>`	begin ”persona”
`      HAMLET`	character data ”HAMLET”
`    </persona>`	end ”persona”
`  </personae>`	end ”personae”
`</play>`	end ”play”

Table 4.1: Events generated by the SAX parser when processing the XML document in the left side of the table.

to data. DOM loads the entire XML document into an object structure, using a parser (actually, often a SAX parser) to build the object structure. It is thus, not possible to traverse and manipulate the object structure until the entire document has been read. Since DOM has to read the entire XML document into the memory to be able to build the tree structure, a large amount of RAM may be required when working with large XML documents. [23, 32]

SAX and DOM

It is difficult to talk of advantages of SAX over DOM or vice versa as the technologies have somewhat different purposes. SAX allows a document to be handled sequentially in a stream-based manner, without having to first read the entire document into memory. SAX is therefore useful when you need a fast, single pass through the XML document to collect relevant data. On the other hand, DOM provides the means for random access manipulation of the data within an XML document, while SAX only provides serial access to them.

4.3.2 XPath

XPath is a specification language designed by the World Wide Web Consortium (W3C) [33]. Being a subset of XQuery, it is an expression language for addressing parts of an XML document, used by various XML technologies such as XSL Transformations (XSLT) and XML Pointer Language (XPointer) (see [27, 34, 35] for a detailed description of XSLT and XPointer).

As the name implies XPath uses path notation for navigating through the hierarchical structure of an XML document [33]. Like DOM, XPath views an XML document as a tree structure consisting of nodes of different type, each representing the entities of a document.

Location path

The primary syntactic construct in XPath is the *location path*, which is a sequence of *location steps* separated by a slash (/). Each step in turn selects a set of nodes relative to the context node and each node in that set is used as a context node for the following step. Consider the following location path:

```
        child::act/child::scene
```

This location path can be evaluated against the XML document in figure 4.5, and will select the `scene` element children of the `act` element children of the document root node (`play`), which is the initial context node.

```
<play>
   <title>
      The Tragedy of Hamlet, Prince of
      Denmark
   </title>
   <act>
      <title>
         ACT III
      </title>
      <scene>
         <title>
            SCENE I.  A room in the castle.
         </title>
         <speech>
            <speaker>
               HAMLET
            </speaker>
            <line>
               To be, or not to be: that is the question:
            </line>
         </speech>
         <speech>
            <speaker>
               OPHELIA
            </speaker>
            <line>
               Good my lord,
            </line>
            <line>
               How does your honour for this many a day?
            </line>
         </speech>
      </scene>
   </act>
</play>
```

Figure 4.5: Simple XML document with root element *play*.

The syntax of a location step consists of an *axis*, a *node test* and zero or more *predicates* [33]. The syntax for a location step is the name of the axis followed by the node test and zero or more expressions each in square brackets. The axis and node test are separated by a double colon (::). A location step thereby has the following syntax:

```
    axis::nodetest[predicate 1][predicate 2]...[predicate N]
```

Axes An axis identifies a direction of traversal. It specifies the relationship between the nodes selected by the location step and the context node. In XPath an axis can be either a forward axis or a reverse axis. An axis which contains the context node and nodes that follow the context node in document order is a forward axis. Document order organises element nodes sequentially after the occurrence of their start-tag in the corresponding XML document. *Child* and *descendant* are examples of forward axes and *ancestor*, *preceding* and *preceding-sibling* are examples of reverse axes.

Node tests A node test specifies the node type and name of the nodes selected by the location step. A node test is true if and only if the node has a name and type equal to the name and type specified by the node test. The :: operator designates which axis the immediately following test should be applied to.

Consider the XML document in figure 4.5, and let the node `play` be the context node. The node test `child::act` will then select all `act` children of `play`. If the context node has no `act` children, it will select an empty set of nodes. It is possible to use the character $*$ as a wild card in the node test. Let `play` be the context node again, then `child::*` will select all children of `play`, in this case the `act` and `title` elements.

Predicates Predicates use equality tests ($=, != , <, >, <=, >=$) to further refine the set of nodes selected by the location step [33]. For instance, the predicate `position()=4` evaluates to true, if the context node is the fourth node in the node-set. To retrieve the first speaker of the XML document in figure 4.5, one could use the following location path:

```
child::act/child::scene/child::speech[position()=1]/child::speaker
```

When evaluated, the above expression will return the element `speaker` with the content “HAMLET”.

Chapter 5

Related work

This chapter presents a survey of related work in the field of peer-to-peer storage and lookup systems. We will explore key features and deficiencies of some prevalent peer-to-peer systems that bear most relevance to the XML Store project. At the end of the chapter some other related technologies are briefly discussed.

The term *peer* is used to describe participants of peer-to-peer systems when these are discussed in general. However, in section 5.1.1 participants are denoted *nodes*, since the routing and location schemes consider participants as abstract entities in an algorithm.

5.1 Peer-to-peer systems

There are currently several peer-to-peer systems in use, and many more are under development. The popularity of peer-to-peer file sharing systems such as Napster [36, 37] and Gnutella [18] has created a flurry of recent research activity into peer-to-peer architectures [1, 2, 3, 4, 17, 38, 39].

Although the exact definition of "peer-to-peer" is debatable, these systems typically lack dedicated, centralised infrastructure, but rather depend on the voluntary participation of peers to contribute resources out of which the infrastructure is constructed (see section 2.3 for a description of peer-to-peer systems). One challenge of peer-to-peer systems is to organise peers into a cooperative, global index (used to map file names to their location in the system) so that all content can be quickly and efficiently located by any peer in the system.

In Napster [36, 37], a central server stores an index of all files available within the system. To retrieve a file, a user queries this central server using the desired file's well known name and obtains the location (the IP address) of a user machine) storing the requested file. The process of locating a file is thus very much centralised and therefore, Napster is not considered a "pure" peer-to-peer system.

Gnutella [18, 40] is a decentralised peer-to-peer system and hence, there are no central servers to query when wishing to locate a file. Instead, peers in the Gnutella network use broadcasting to a great extent to locate files.

Peer-to-peer systems such as the present XML Store, CFS [1, 2], PAST [3, 4], OceanStore [38, 39] and Freenet [17], seek to provide highly available storage with efficient location-independent routing and location in a decentralised environment. The notion *routing* refers to the forwarding of messages (queries) from one peer to another in the peer-to-peer system, until the destination peer is reached.

The core of many of these peer-to-peer systems consists of novel distributed algorithms for routing and locating data independently of its physical location. These algorithms resemble distributed hash tables, supporting the basic operations insert, delete and lookup of ⟨key,value⟩ pairs. In essence, they distribute ⟨key,value⟩ pairs across various peers in a large network, in a manner that facilitates scalable access to these pairs using the key. [41]

5.1.1 Decentralised routing and location

This section discusses some decentralised routing and location schemes that are used by peer-to-peer systems related to the XML Store system.

The Chord routing and location scheme [21, 42, 43, 44] constitutes the core of the XML Store system, as well as of the CFS system [1, 2]. Chord is a peer-to-peer routing substrate, which is efficient, scalable, fault-tolerant and self-organising. The Chord protocol will be discussed in detail in chapter 8 – the following is only a brief presentation of Chord and has been included for comparison reasons.

With Chord, each node and each data item is assigned a unique identifier. Nodes are ordered according to their identifiers, in a one-dimensional circle corresponding to the Chord identifier space. Data items are assigned to Chord nodes in a location-independent manner. Each Chord node uses a logarithmic-sized routing table ($O(\log N)$), called a *finger table*, to route and locate data in the Chord system. The routing and location time of Chord is $O(\log N)$, where N is the number of nodes in the system.

Several recent projects concerning decentralised routing and location are closely related to Chord [21, 42, 43, 44]. Some of these projects are Plaxton [45], Pastry [22], CAN [46] and Tapestry [47].

The Plaxton scheme [45] is innovative in that routing and location of data can be achieved across an arbitrarily-sized network, while using a small constant-sized routing information at each hop. With Plaxton, nodes and data are assigned identifiers which are represented by a common base b. Each Plaxton node uses a local routing table (called a *neighbour map*), to incrementally route messages to the destination identifier, digit by digit. This routing method guarantees that any existing node in an N-node Plaxton system will be found within at most $O(\log N)$ hops [45]. The location of a specific data item d, involves routing towards a so called *root node* of d, which stores information about the location of d. If information about the location of d is encountered at a node on the way to the *root node*, then the message is immediately redirected to the node containing d.

There are some limitations related to the Plaxton scheme. First, Plaxton uses *global knowledge* to deterministically choose a *root node* to which a given identifier for some data should be mapped. This obviously complicates dynamic additions and removals of nodes [47]. The *root node* is essential for location and therefore represent a single point-of-failure to the Plaxton system [47]. Secondly, the Plaxton scheme makes an assumption that the Plaxton network is a static data structure, which means that dynamic node insertions, deletions and failures cannot be handled. Therefore, the Plaxton scheme is not considered fully self-organising.

The routing and location mechanisms of Tapestry [47] (used by OceanStore) and Pastry [22] (used by PAST) are similar to the routing and location mechanisms introduced in the Plaxton scheme [45]. The approach of routing towards nodes that share successively longer address (identifier) prefixes, is common to all three schemes [22]. However, the location mechanisms differ in that neither Pastry nor Tapestry have a single *root node* per data item storing the location of it. Instead, this information is stored at multiple nodes, thereby avoiding a single point-of-failure.

In both Pastry and Tapestry, the expected number of routing hops is $O(\log N)$, where N is the number of live nodes in the network. The size of the routing table, which is maintained in each node, is $O(\log N)$. [22]

CAN (*Content-Addressable Network*) [46] routes messages in a d-dimensional space. In CAN, nodes are addressed by their IP addresses. Each data record has a unique key which is hashed so that it corresponds to a point in the d-dimensional space. Each CAN node maintains a routing table with $O(d)$ entries. Any node can be reached in $O(dN^{1/d})$ routing hops. Unlike the routing tables of Chord, Pastry and Tapestry, the routing table of CAN does not grow with the network size. On the other hand, the number of routing hops grows faster than $O(logN)$. However, if d is chosen to be $d = (\log N)$ CAN will achieve the same routing and location complexity as Chord, Pastry, Tapestry and Plaxton.

Although the Chord protocol is closely related to the Plaxton, Pastry and Tapestry protocols, there is a difference in their routing approach. Instead of routing towards nodes that share successively longer address prefixes with the destination, as Pastry, Tapestry and Plaxton do, Chord forwards messages based on numerical difference with the destination address [22].

Deterministic location

Deterministic location means that the system guarantees to find data if they exist in the system.

Peer-to-peer systems like Gnutella and FreeNet are not guaranteed to find an existing file [17, 18]. If a file is not returned when a user looks it up, she will not have any idea about whether the lookup has failed because the file does not exist in the system or because the file was simply not found.

Chord, Plaxton, Pastry, Tapestry and CAN represent a second generation of peer-to-peer location and routing schemes which, unlike Gnutella and Freenet, guarantees a definite answer to a query in a bounded number of network hops, while retaining scalability and self-organisation [3, 22, 43, 45, 46, 47].

Network locality

Retrieving data from a nearby node, in terms of some proximity metric (such as the ping delay, the number of IP routing hops or geographic distance), minimises latency and network load [22]. It is therefore desirable to direct a lookup query towards a node that is located relatively near the client making the request.

The Chord protocol makes no explicit effort to take network locality into account when routing data or queries[1] [21, 42, 43, 44].

In contrast, Plaxton, Tapestry and Pastry consider network topology when routing data, thereby seeking to minimise the distance that data and queries travel according to some scalar proximity metric [22].

Like Chord, CAN does not attempt to approximate real network distance [46].

Self-organisation

Operating under continuous changes, the routing and location infrastructure must be able to adapt to a dynamic changing environment where nodes come and go. Thus, the topology of the location and routing infrastructure must be self-organising [47].

Chord is self-organising in that it automatically adapts to the arrival, departure and failure of nodes. Like Chord, Pastry, Tapestry and CAN posses self-organising properties as they support dynamic node insertions and deletions and are able to handle node failures [22, 43, 46, 47, 48]. The Plaxton scheme, on the other hand, is not fully self-organising, as it cannot adapt to dynamic node insertions, deletions and failures [45].

Fault-tolerance

Typically, arbitrary node failures must be tolerated in peer-to-peer systems. The basic Plaxton scheme [45] is sensitive to a variety of failures. Since each file has a single *root*, a single point-of-failure for each piece of data exists. This single point of failure is a potential subject of denial of service attacks and constitutes an availability problem.

Chord, Pastry, Tapestry and CAN are deterministic, and thus vulnerable to malicious or failed nodes along the route that accept messages but do not correctly forward them. This means that repeated queries could thus fail each time, since they are likely to take the same route. To overcome this problem, the routing is randomised in Pastry [22].

Summary of routing and location

Table 5.1 gives an overview of some of the most important characteristics of the aforementioned routing and location schemes. As mentioned earlier, the Chord protocol will be discussed in detail in chapter 8.

[1]CFS, which is built on top of Chord, use network locality when routing data.

	Chord	**Pastry**	**Tapestry**	**CAN**	**Plaxton**
Lookup (hops)	$O(\log N)$	$O(\log N)$	$O(\log N)$	$O(dN^{1/d})$	$O(\log N)$
Routing info per node	$O(\log N)$	$O(\log N)$	$O(\log N)$	$O(d)$	$O(\log N)$
Deterministic location	Yes	Yes	Yes	Yes	Yes
Scalable	Yes	Yes	Yes	Yes	Yes
Self-organising	Yes	Yes	Yes	Yes	No
Decentralised	Yes	Yes	Yes	Yes	Yes
Network locality	No	Yes	Yes	No	Yes

Table 5.1: Characteristics of related peer-to-peer routing and location schemes. N is the number of nodes in the system in question and d is the number of dimensions in the CAN scheme.

5.1.2 Properties and features of related peer-to-peer systems

In the previous section, we described different decentralised routing and location schemes used by various peer-to-peer systems, related to the XML Store system. In this section we describe some of the properties and features of related peer-to-peer systems. The peer-to-peer systems in question are: CFS, PAST, Napster, Gnutella, OceanStore and Freenet.

Scalability

The scalability of data location and query in peer-to-peer systems is of paramount concern. It should be possible to extend the system with new resources at a reasonable cost and there should be no performance bottle necks.

Napster [36, 49] is not considered scalable, since this system relies on a central server to locate files and this results in a performance bottleneck. This makes Napster vulnerable since there is a single point-of-failure and it is expensive to scale the central server. Gnutella [18] displays a limited scalability because of the extensive use of a broadcasting. The broadcast based protocol also incurs high bandwidth requirements.

The XML Store, CFS, PAST, FreeNet and OceanStore all scale well, due to the routing and location schemes that they use.

Availability

The availability of a system and of data, is a measure of the proportion of time that it is available for use [5].

Freenet is not intended to guarantee permanent file storage – it is hoped that a sufficient number of peers will join with enough storage capacity so that most files will be able to remain indefinitely [17].

In OceanStore, CFS and PAST, data are replicated and stored on multiple peers. This replication provides availability in the presence of network partitions and durability against failure and attack. A given replica is independent of the peer on which it resides at any one time. [2, 3, 39]

Presently, the XML Store system does not replicate data, which means that the degree of availability may not be very high.

Load balancing

Load balancing refers to the problem of spreading data (and requests) evenly among peers, according to the capacity of the peers.

The Plaxton scheme has good load distribution properties, since identifiers of data are randomly mapped throughout the infrastructure [45].

Storage of data in CFS is block-oriented – files are split into blocks of a fixed size. In the XML Store system, files are split into fragments of varying size according to the inherent tree structure of the documents. In both CFS and the XML Store, files are therefore distributed across multiple peers, which increases load balancing, as popular files are spread across multiple peers, and not placed on a single hotspot.

PAST, on the other hand, stores whole files. Compared to PAST, the storage strategies used in CFS and the XML Store, increases file retrieval overhead, since each file block/fragment must be located by using a separate Chord lookup operation. On the other hand, the storage strategy permits parallel block/fragment retrievals, which benefits large files. Another advantage of splitting up files is that files that are too large to be stored at any one peer can be stored as blocks/fragments if the system as a whole has enough free space [2].

To accommodate peers with heterogenous storage space, CFS introduces *virtual servers*, thereby hosting multiple logical peers per physical peer [2]. PAST uses a similar strategy, but to a lesser extent [3].

Anonymity

An important issue in peer-to-peer systems, and particularly in storage and file-sharing systems, is privacy and anonymity [4]. A provider of storage space used by others may not want to risk prosecution for content it stores, and clients inserting or retrieving files may not wish to reveal their identity [4].

Peer-to-peer systems lend themselves naturally to anonymous communication, as there is no central point at which to collate information about users. However, careful design choices must be made to ensure anonymity.

One of the main design goals of Freenet is to provide anonymity for both authors and readers of data, i.e. provide a file system that allows files to be inserted, stored and requested anonymously [17]. Freenet achieves anonymity by exchanging data via a chain of peers, where no peer except for the recipient is aware of where the chain terminates. However, the anonymity of Freenet comes at a price as it limits performance of the system. [2]

Systems like Gnutella, on the other hand, fail to achieve anonymity as peers connect directly to exchange data, thus revealing their addresses to each other.

The XML Store and CFS do not attempt to provide anonymity. The focal points of these systems are instead efficiency and robustness [2].

In PAST, users do not need to reveal their identity and the files they are retrieving, inserting or storing [4]. Each user holds a public key, which is not easily linkable to the user's identity, unless the user voluntarily reveals the binding [4].

Searching

Although hash functions can help place and locate content deterministically, they lack the flexibility of searching, which is a useful operation to find content without prior knowledge of exact file names.

Both Gnutella and Napster provide search facilities [18, 36, 49]. They present a search interface to clients, rather than retrieving uniquely identified data, making them more like search engines than distributed hash tables. However, Gnutella searches are based on broadcast which is clearly inefficient and Napster searches are based on a central server which constitutes a single point-of-failure.

Neither the XML Store nor CFS and PAST provide facilities for searching [2, 3]. However, Dabek et al. [1] are currently working on making a scalable distributed search facility for CFS.

Summary of peer-to-peer systems

Table 5.2 gives an overview of the most important properties and features of the discussed peer-to-peer systems CFS, PAST, OceanStore, Freenet, Napster, Gnutella and the XML Store.

5.2 Other related technologies

5.2.1 XML databases

Tamino XML Server and Apache Xindice are both centralised, native XML databases. The benefit of storing XML documents in a native XML database compared to storing them in a relational database is that no conversion of data formats is necessary. Data are inserted as XML and retrieved as XML. A schema independent model employed by both Tamino and Xindice makes it possible to store very complex XML structures that would be difficult or impossible to map to a more structured database. [50, 51]

	XML Store	**CFS**	**PAST**	**Ocean-Store**	**Freenet**	**Napster**	**Gnutella**
Routing & location	Chord	Chord	Pastry	Tapestry	Internal	Central server	Broad-cast
Deterministic location	Yes	Yes	Yes	Yes	No	No	No
Scalable	Yes	Yes	Yes	Yes	Yes	No	Limited
Decentralised	Yes	Yes	Yes	Yes	Yes	No	Yes
Replication	Possible	Yes	Yes	Yes	Yes	No	No
Caching	Yes	Yes	Yes	Yes	Yes	No	No
Split up files	Yes	Yes	No	No	No	No	No
Search facility	No	No	No	No	No	Yes	Yes
Immutable data	Yes	Yes	Yes	No	–	No	No

Table 5.2: Features and properties of related peer-to-peer systems. "–" means that no information about this property was available.

Both applications provide XPath-based query capabilities in combination with indexing. Tamino also offers text retrieval for searching in non-indexed parts of an XML document and will support XQuery when it is ratified as a W3C standard.

Since none of the two solutions are distributed, and since queries are not part of our focus, they will not be discussed further in this thesis.

5.2.2 Peer-to-peer framework

JXTA is a network programming and computing framework for peer-to-peer application development [52]. It defines a set of open protocols, that use XML-encoded messages. JXTA provides protocols and services to implement a full peer-to-peer application.

To build a peer-to-peer system, JXTA needs an underlying routing and location scheme, such as Chord. An ongoing project *JXTA Distributed Indexing* seeks to combine Chord and JXTA to provide a distributed index service, which can be used to index and search content [52].

We have chosen not to build the XML Store on top of JXTA, since we find that an implementation of the aforementioned protocols is too comprehensive and restrictive for our purpose.

Part II

Analysis & design

Chapter 6

XML Store overview

This section provides an overview of the components comprising the XML Store, and describes the interaction between the parts of the system as well as the interaction between an external application and the XML Store system.

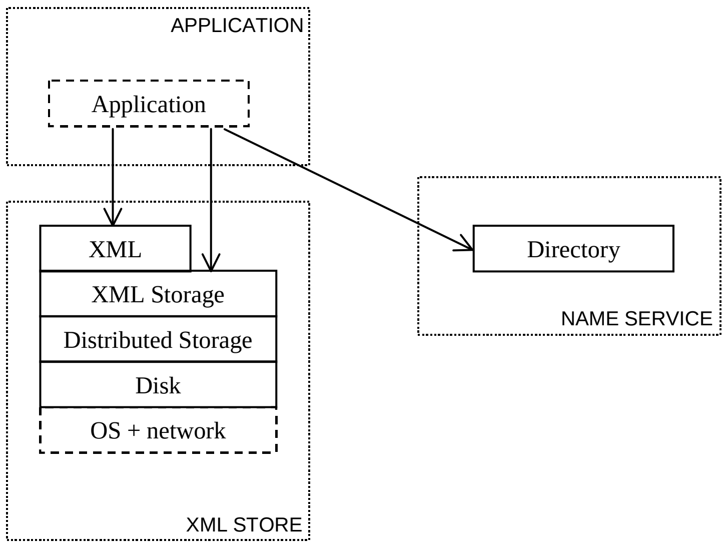


Figure 6.1: The components of an application utilising the XML Store system and the four layers of the XML Store component.

Figure 6.1 shows an application utilising the XML Store. There are three main components, namely Application, XML Store and Name Service. The XML Store component provides an API for processing and manipulating XML documents, and offers storage and retrieval of these documents as well. When storing a given document the XML Store returns a value reference referring to the document. This reference can either be stored locally by the application or be associated with a symbolic name using the Name Service. Other applications interested in the document can obtain the value reference for a document from the Name Service.

Layer	**Example Operations**
XML	Various methods for accessing the children of an XML element, accessing the content of a character data node, removing and adding children.
XML Storage	`ValueReference save(Node)` `Node load(ValueReference)`
Distributed Storage	`save(byte[],ValueReference)` `byte[]load(ValueReference)`
Disk	`save(byte[],ValueReference)` `byte[] load(ValueReference)` `delete(ValueReference)`

Table 6.1: Operations of the layers in the XML Store component.

The XML Store can conceptually be considered as consisting of four layers, each responsible for a clearly defined set of tasks, as shown in 6.1. The *XML* layer models XML documents as an object structure, allowing the programmer to work with an abstract representation of the document instead of a flat, sequential text file. The API allows traversal of documents and various methods for modifying the document, e.g. removing an element, adding an element, replacing an element etc. The API is value-oriented, so all modifications to a document will result in a new document containing the changes, with the old document still intact. It also lets the programmer externalise the document to the typical flat text file format, to facilitate communication with systems that expect this format.

The *XML Storage* layer implements the storage strategy. It is responsible for converting documents from the XML layer to sequences of bytes that the underlying Distributed Storage layer can understand. The layer converts each node in an XML document to a byte sequence, and stores the node. When storing a document a value reference to the document is returned to the application. A document can only be loaded if you know its value reference.

The *Distributed Storage* layer is a peer-to-peer system based on the Chord protocol [21, 42, 43, 44]. It can be regarded as a distributed hash table that allows storing and retrieval of values as sequences of bytes associated with a (hash) key in the form of a value reference. The peer (machine), on which the value is actually stored, is not known by higher layers.

When the Distributed Storage layer has located the peer responsible for storing a particular value, it stores the value on that peer using the *Disk* layer. The Disk layer simply persists ⟨byte sequence, value reference⟩ pairs to the local file system for permanent storage.

Name Service

The *Name Service* is a simple service mapping symbolic names to value references. It is completely separate from the XML store, as the XML Store only knows of values and value references and not of any symbolic names.

Symbolic names can be updated to point to new value references, as it is necessary to have some mechanism for publishing the current version of some document. To reflect updates we need some way of accessing public and updateable names for the documents that are published. This is discussed in chapter 10.

Summary

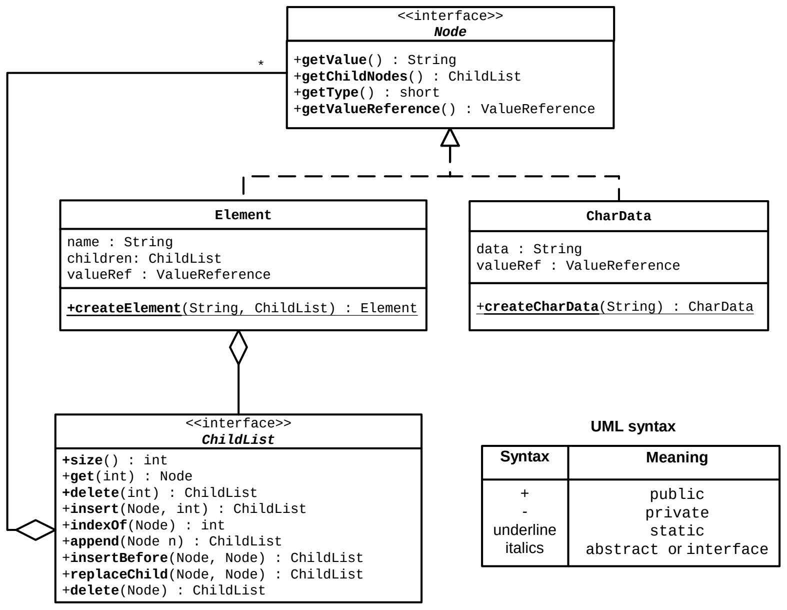
We have now presented an overview of the components comprising the XML Store. A more detailed discussion of the components can be found in chapter 7 (API design), chapter 8 (The Chord protocol) and chapter 9 (Value-oriented storage).

Chapter 7

API design

In this chapter we describe the value-oriented XML API that has been designed and implemented for the XML Store. Our goal is to provide a simple and easy-to-use API that supports value-oriented programming and has the same strength and applicability as DOM.

We start the chapter by giving a description of the class structure of an XML document and of the methods in the `ChildList` class. We then describe our adaptation of XPath, which is used in utility methods for modifying whole XML documents. Examples of the use of these methods are also given. Finally we present an analysis of different ways of representing a child list.



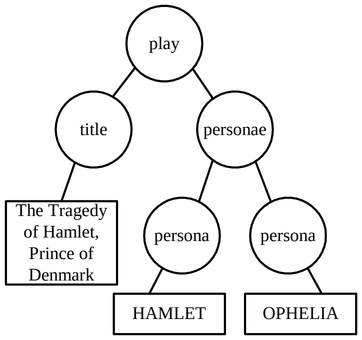
<<interface>>
Node
+getValue() : String
+getChildNodes() : ChildList
+getType() : short
+getValueReference() : ValueReference
*
Element
name : String
children: ChildList
valueRef : ValueReference
+createElement(String, ChildList) : Element
CharData
data : String
valueRef : ValueReference
+createCharData(String) : CharData
<<interface>>
ChildList
+size() : int
+get(int) : Node
+delete(int) : ChildList
+insert(Node, int) : ChildList
+indexOf(Node) : int
+append(Node n) : ChildList
+insertBefore(Node, Node) : ChildList
+replaceChild(Node, Node) : ChildList
+delete(Node) : ChildList
UML syntax

Syntax	Meaning
+	public
-	private
underline	static
italics	abstract or interface

Figure 7.1: The class structure of an XML document as represented in the XML Store.

7.1 Class structure

The proposed API provides a tree-structured abstraction of XML data. An XML document is represented by the class structure shown in figure 7.1. Since we use the Minimal XML (see section 4.2) syntax for XML documents only element and character data nodes are represented.



```
Node ch1 = createCharData("HAMLET");
Node ch2 = createCharData("OPHELIA");
Node ch3 = createCharData("The Tragedy of Hamlet, " +
                                   "Prince of Denmark");
Node persona1 = createElement("persona", new Node[] {ch1});
Node persona2 = createElement("persona", new Node[] {ch2});
Node title = createElement("title", new Node[] {ch3});
Node personae = createElement("personae", new Node[]
                                   {persona1, persona2});
Node play = createElement("play", new Node[] {title, personae});
```

Figure 7.2: Tree representation of the XML document and the Java code necessary to create this document.

A tree representation of an XML document can be constructed by the operations `createElement()` and `createCharData()` and de-constructed or traversed by `getChildNodes()` and `getValue()`. Figure 7.2 shows a tree representation of an XML document and the operations necessary for constructing this document.

Instances of the classes `CharData` and `Element` can only be created by using `Element.createElement()` and `CharData.createCharData()`. This allows us to employ hashed consing which ensures that there do not exist two trees or DAGs in the object structure whose tree expansions are isomorphic, thereby ensuring maximal sharing.

By employing the strategy shown in figure 7.2 all documents can be created with `createElement()` and `createCharData()`. In addition, all forms of manipulations of a document can be performed using these two operations combined with the de-construction methods `getChildNodes` and `getValue()`. A modification of a document is accomplished by manually traversing the node structure, modifying nodes and building a new tree accordingly, as explained in chapter 3. Modifying a large XML document this way is cumbersome, though. We therefore provide the programmer with methods that ease the modification of XML documents. We first present the methods of the `ChildList` class.

7.2 Methods for manipulation of child nodes

The methods in the `ChildList` class provides the means to manipulate the child nodes of an element. Table 7.1 shows the core methods, which are methods from which all other relevant methods can be created.

Methods	**Description**
`int size()`	Returns the number of child nodes.
`Node get(int index)`	Returns the `Node` at position `index`.
`ChildList insert(Node n, int index)`	Returns a new `ChildList` where `n` has been inserted at position `index`.
`ChildList delete(int index)`	Returns a new `ChildList` where the node at position `index` has been deleted.
`int indexOf(Node n)`	Returns the position of node `n` in the `ChildList`.

Table 7.1: The core methods of the `ChildList` interface.

By combining the core operations shown in table 7.1 it is possible to construct more advanced methods. For instance the Java code for the derived method `insertBefore()` is:

```
ChildList insertBefore(Node newChild, Node refChild) {
  return insert(newChild, indexOf(refChild));
}
```

The rest of the derived methods of the `ChildList` interface are shown in table 7.2. Notice that all methods in the `ChildList` interface are value-oriented and therefore result in a new child list containing the changes with the old child list still intact.

Methods	**Description**
`ChildList append(Node n)`	Returns a new `ChildList` where `n` has been appended to the end of the child list.
`ChildList delete(Node n)`	Returns a new `ChildList` where node `n` has been deleted.
`ChildList replaceChild(Node newChild, Node oldChild)`	Returns a new `ChildList` where `newChild` is replaced by `oldChild`.
`ChildList insertBefore(Node newChild, Node refChild)`	Returns a new `ChildList` where `newChild` has been inserted before `refChild` in the `ChildList`.

Table 7.2: The derived methods of the `ChildList` interface.

The applicability of the methods in table 7.2 are somewhat limited since they only offer possibilities for manipulation of the children as single nodes. In order to provide an API that meets our requirements we offer methods that can manipulate an entire document accessed through the root of the document. These utility methods recursively descent the tree structure and build a new document beginning from the root element. When they reach the part of the document that should be modified they use the operations in table 7.1 to manipulate this part of the document. The utility methods use `createElement()` and `createCharData()` to build the new document, thereby ensuring maximal sharing.

Since the utility methods build the new document starting from the root element, we need some sort of path or pattern to specify the navigation from the root of the document to the node(s) that we want to modify. We choose to use a simplified and abbreviated syntax of the *location path* expression of XPath, explained in section 4.3, to locate the desired nodes.

7.3 Our adaption of XPath

As described in chapter 4.3 the location path of XPath is a sequence of location steps separated by a slash (/). A location step has the following syntax:

```
axis::nodetest[predicate 1][predicate 2]...[predicate N]
```

When a location path is evaluated each location step in turn selects a set of nodes relative to the context node and each node in that set is used as a context node for the following step. An example of a location path is:

```
child::personae/child::persona[position()=4]
```

For convenience we have chosen to implement an abbreviated syntax of the location path that allows common cases to be expressed concisely. Only element nodes have tag names in our object structure, so we only consider element nodes when evaluating a location path. Since we make use of sharing of subtrees, parents and siblings are not uniquely determined. Therefore, we only have the *child* axis, and consequently `child::` can be omitted from the location step. For example, a location path `act/scene` is short for `child::act/child::scene`. Finally, we only have one kind of *predicate*, namely the `position()` operation. Hence, the text “`position()=`” can be omitted from the *predicate* part of the location step, i.e. `act[3]` is equivalent to `act[position()=3]`.

Here are some more examples of the abbreviated syntax:

- `*/act` selects all `act` grandchildren of the context node.
- `act/scene[5]/line[2]` selects the second line of the fifth scene of the act child nodes of the context node.

7.4 Utility methods for manipulation of XML documents

The utility methods for modifying whole XML documents are shown in table 7.3. Notice that modification applies to *all* subtrees that match the XPath expression.

The methods in table 7.3 correspond to a minimal subset of the methods offered by DOM (refer to Le Hors et al. [53] for a full overview of a current DOM implementation). However, the DOM API is imperative, which means that nodes in the tree are updated destructively. In contrast we provide a value-oriented API, which, besides being both simple and as applicable as DOM, allows sharing of identical subdocuments, eliminates the need of locking mechanisms and forms the basis of secure and efficient replication and coalescing as well as preserving all previous versions of a modified document.

Methods	**Description**
`ChildList lookup(Node root, String path)`	Returns a `ChildList` from the document specified by `root` that matches the abbreviated XPath expression specified by `path`.
`Node append(Node root, String path, Node newNode)`	Returns a new document where `newNode` has been appended to the end of the list of children of the elements specified by the abbreviated XPath expression `path`.
`Node delete(Node root, String path, Node oldNode)`	Returns a new document where `oldNode` has been deleted from the list of children of the elements specified by the abbreviated XPath expression `path`.
`Node insertBefore(Node root, String path, Node refNode, Node newNode)`	Returns a new document where `newNode` has been inserted before `refNode` in the list of children of the elements specified by the abbreviated XPath expression `path`.
`Node replace(Node root, String path, Node oldNode, Node newNode)`	Returns a new document where `oldNode` has been replaced by `newNode` in the list of children of the elements specified by the abbreviated XPath expression `path`.

Table 7.3: Methods for manipulation of an XML document.

7.4.1 Examples of uses of the API

Consider the XML document A in figure 7.3 and assume that we have loaded the root element *play* into the variable `play`. Then `lookup(play, "personae/persona[2]")` will return a `ChildList` with one element, namely the `persona` element that has the character node with contents “OPHELIA” as child (notice that the first position in an XPath predicate is 1).

A new `persona` element can be inserted before the above found `persona` element (which we keep in the variable `ophelia`) by executing the following operation: `insertBefore(play, "personae", ophelia, horatio)`. The resulting document A' is also shown in figure 7.3. As a final example we remove the newly inserted element: `remove(play, "personae/persona[2]")`. This results in a document with a tree representation that is isomorphic to the tree representation of the original document A in figure 7.3.

7.5 Representing child lists

When working with an element in an XML document we need to have a way of accessing and modifying the child nodes of the element. An XML element may have zero or more children, consisting of either nested elements or character data blocks. The children of an element are ordered sequentially so it is natural to think

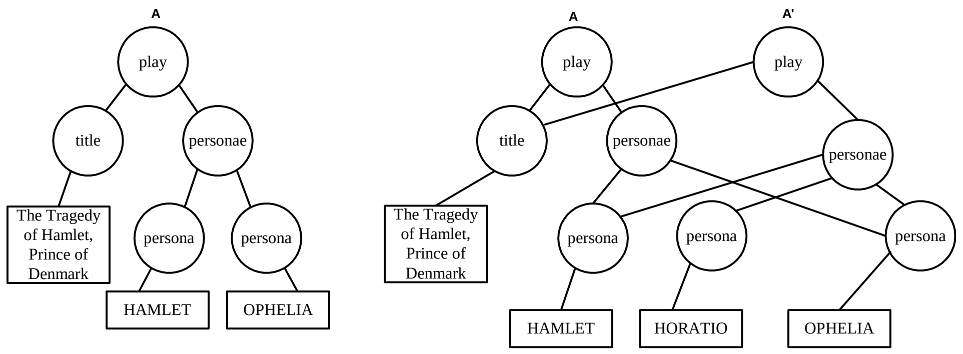


Figure 7.3: Tree representation of an XML document A and the result A' of the operation `insertBefore(play, "personae", ophelia, horatio)` performed on A.

of them in terms of a list or an indexed array when deciding how to represent them in the object structure. We have decided to give the user the impression that she is working with a random access list. As described above the core operations that must be supported are: `get(index)`, `delete(index)`, `insert(node, index)`, `size()` and `indexOf(node)`.

These operations are essential as they are used frequently when processing an XML document, e.g. traversing the structure and accessing the contents of the document or modifying the document. The operations must be implemented efficiently for the proposed API to be useful for working with XML documents.

In this section we focus only on implementing the core operations as the derived operations depend solely on the implementation of the core operations.

We have based the API on the assumption that the user processes the document one node at a time. Consider the case of a user wanting to create an XML element having five children. The user constructs the child list by inserting the five children one at a time. As we work value-oriented a child list cannot be altered – whenever a change has been made the result is a new child list reflecting the change with the old one still intact. In the example, in addition to the wanted child list `[1,2,3,4,5]`, the intermediate child lists `[1]`, `[1,2]`, `[1,2,3]`, `[1,2,3,4]` are also built.

There are several ways to implement the child list and each has its advantages and drawbacks. In the following we present two different solutions. First a simple solution based on the use of arrays is sketched. It is easy to understand but has severe drawbacks in performance. Next we present a solution based on persistent, balanced trees that offer acceptable performance even when processing elements with a large number of children. Both solutions are implemented in the prototype of the XML Store.

We examine the runtime complexity of the core operations in the suggested solutions and assess the time and space consumption for the scenario of building a child list with n elements, one element at a time.

7.5.1 Array solution

In the first solution children are simply stored in an array according to their order. This obviously makes the `get(index)` operation very fast ($O(1)$) as this is merely a question of accessing the requested position in the array. As the size of the array is known when it is created and cannot be altered, the `size()` operation can easily be accomplished in constant time.

The operations that modify a child list are unfortunately very slow, when implemented with arrays. Consider the case of an element e being inserted into a list $xs1$ producing a new list $xs2$. A new array of size `(xs1.size() + 1)` has to be created and each child (or more precisely: a reference to each child) in $xs1$ has to be copied to $xs2$ along with the newly inserted child. This obviously takes time proportional to the number of children (n) in the child list being processed, yielding a complexity of $O(n)$. Deletion is equally demanding requiring all children except the deleted one to be copied to a newly created array of size `(xs1.size() - 1)`.

Finally, the `indexOf()` operation requires a linear search of the array, resulting in a run time of $O(n)$. Figure 7.4 summarises the runtime complexities of the core operations in the array based solution.

Operation	**Runtime Complexity**
`Node get(int index)`	$O(1)$
`ChildList delete(int index)`	$O(n)$
`ChildList insert(Node node, int index)`	$O(n)$
`int size()`	$O(1)$
`int indexOf(Node node)`	$O(n)$

Table 7.4: Runtime complexity of the core operations for the array based solution.

Now consider the scenario of building a child list with n elements. For the intermediate child lists we have to create a total of n arrays of size $(1, 2, ..., n)$. This leads to a $\Theta(n^2)$ space consumption [54]. The time spent is also quadratic as we have n operations of time $(1, 2, ..., n) = \sum_{i=1}^{n} i = \Theta(n^2)$. Though accessing elements is fast it does not warrant the poor performance when adding or deleting elements. Therefore, the solution is obviously inadequate for processing large XML documents.

7.5.2 Binary tree solution

The main problem with the former solution is that for each modification of the child list, all or almost all children must be copied. In this solution we consider representing the children in a value-oriented binary tree, enabling us to share unchanged parts of the child list when a modification is carried out. We still present a list interface to the application programmer, but underneath we maintain a tree structure, where the order of the children is reflected by their position in the tree. To accomplish this, we consider the tree a binary tree that satisfies following property:

> Let x be a node in a binary tree. If y is a node in the left subtree of x, then y *precedes* x. If y is a node in the right subtree of x, then x *precedes* y (adapted from Cormen et al. [54]).

This implies that an in-order traversal of the tree will retrieve the children in ascending order. As subtrees can be shared, a tree node can reflect one position in one child list and another position in the context of another child list. This is shown in figure 7.4 where one tree represents two different child lists. Node E is child number five in one child list and child number one in the other. Observe that this means that although we consider the tree to be a binary search tree, we cannot store any keys. The position of a child in the child list must be determined solely by its position in the tree, not by storing indices in the tree nodes.

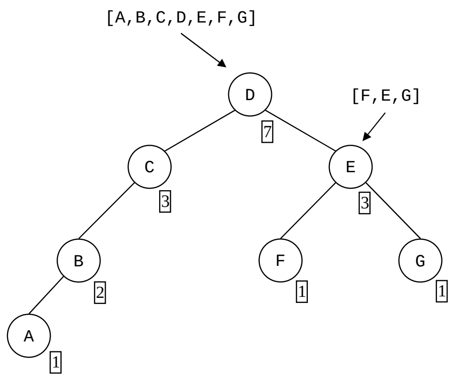


Figure 7.4: Two child lists and their tree representation. Boxes show the size of each sub tree.

We still need a way to locate node number i in a given tree, though. By storing the size of the subtree in each tree node it is possible to determine the order of the tree nodes (as well as implementing the `size()` operation in constant time). Figure 7.5 shows a simple recursive algorithm, `OS-Select`, for finding the i'th element in a tree. `OS-Select` is adapted from Cormen et al. [54] that use it for order statistic operations on dynamic sets. Using this algorithm it is easy to implement the `get(index)` operation in time $O(h)$, where h is the height of the tree.

We have now shown how binary trees can be used for representing ordered child lists and how identical sublists can be shared by sharing subtrees in the tree structure. It is quite simple to implement the `insert()` and `delete()` operations in a regular binary tree. However, we risk severely unbalancing the tree resulting in a worst case performance similar to the array solution.

To prevent this, we propose a solution based on value-oriented red-black trees. Red-black trees are an elegant near-balanced binary tree scheme that guarantees

```
OS-Select (x, i)
    r = x.left.size + 1;
    if (i == r)
        return x
    elseif i < r
        return OS-Select(x.left, i)
    else
        return OS-Select(x.right, i - r)
```

Figure 7.5: Pseudo code for finding the i'th element in a tree with root x. The code is adapted from Cormen et al. [54].

$O(\log n)$ worst-case running time of operations such as `delete()` and `insert()`. Red-black trees are *almost* balanced, in that a red-black tree with n internal nodes has height at most $2\log(n+1)$ [54]. A red-black tree must satisfy the red-black properties [54]:

1. Every node is either red or black.
2. Every leaf is black.
3. If a node is red, then both its children are black.
4. Every simple path from a node to a descendant leaf contains the same number of black nodes.

Okasaki [55] has described the implementation of red-black trees in a functional and thereby value-oriented setting. We base our implementation on his algorithm as well as on extensions presented by Kahrs [56].

Whenever a modification is made to a value-oriented binary tree a number of new nodes have to be created to build the resulting tree, as described in chapter 3. The new nodes that have to be created, correspond to the nodes on the path from the root of the tree to the place where the modification took place. Worst case $O(h)$ new nodes have to be created, where h is the height of the tree.

When inserting into or deleting a node from a red-black tree, balancing operations are carried out to maintain the tree's red-black invariants and thereby keeping the tree balanced. The balancing operations affect the number of nodes that have to be created when inserting or deleting and thereby also the time and space complexity of the algorithm on average. In our value-oriented version of the red-black tree, only a constant number of new nodes are created in each balancing operation, and the balancing operation is only executed on the nodes on the path from the modified node to the root of the tree. Most of the time subtrees are simply shared without the need for creation of new nodes reflecting changes in the tree. So even though balancing introduces more changes to the tree than modification of an ordinary binary tree, it still does so to a limited degree. The number of newly created nodes is still proportional to $\log n$ when modifying a red-black tree, as thus does not affect the overall runtime complexity.

When implementing the child list in the described way the modification operations can be accomplished in time $O(\log n)$, as summarised in figure 7.5. The `indexOf()` operation requires a search of all nodes in the tree leading to a linear runtime in the worst case.

Operation	**Runtime Complexity**
`Node get(int index)`	$O(\log n)$
`ChildList delete(int index)`	$O(\log n)$
`ChildList insert(Node node, int index)`	$O(\log n)$
`int size()`	$O(1)$
`int indexOf(Node node)`	$O(n)$

Table 7.5: Runtime complexity for the child list solution based on red-black trees.

Returning to the scenario of building a child list with n elements, we obtain the following space consumption. Each of the n operations creates new nodes proportional to $\log n$ resulting in a total space consumption of $O(n \log n)$. The runtime complexity of building the child list is also $O(n \log n)$ due to having n operations of $O(\log n)$ time.

The only operation that is not satisfactory in any of the solutions is the `indexOf()` operation. It cannot be improved from linear running time unless an alternate data structure, e.g. a hash table mapping value references to positions, is maintained for each child list. However, since hash tables are based on arrays, the `delete(index)` and `insert(node, index)` operations will have running times similar to the array based solution (see table 7.4), if indexes are needed for every child-list. Alternatively, indexes could be build for selected child lists, e.g. when the creation of the child list is finished or the first time the `indexOf()` is called.

7.6 Summary

We have presented a way of working with and especially manipulating XML documents in a value-oriented environment. We have shown how the proposed operations can be implemented with acceptable performance and we maintain the advantages of working value-oriented in all respects. The proposed API can compete with DOM for applicability and its runtime properties regarding space as well as time consumption are acceptable for practical use even with very large documents.

Chapter 8

The Chord protocol

When designing a distributed XML Store based on a peer-to-peer network we need to address the problem of efficiently locating the peer which is either going to be responsible for a particular piece of data or which is already responsible for the data. Due to the decentralised nature of a peer-to-peer system, each peer cannot be aware of all other peers in the system. Consequently a strategy for finding a given peer based on some criteria is needed.

A number of algorithms have been suggested for solving this problem, as described in section 5. One of the most promising is Chord, recently presented by Stoica et al. [42]. Chord seems well suited for the task of providing distributed storage for the XML Store, as it combines scalability and simplicity and furthermore is well documented. Note that the Chord protocol uses the term *node* to denote a peer in the system.

The chapter is structured as follows: Initially the Chord protocol is described and analysed. The focal points will be the basic Chord operations: How are they carried out, their properties and how are they accomplished when having limited knowledge of the other nodes in the system. Then caching, load balancing, fault tolerance and server selection are described. These techniques are not an indispensable part of the Chord protocol, but have been included in the CFS system [2] which is based on the Chord protocol. We conclude this chapter by evaluating the usefulness of the Chord protocol in the context of the XML Store system.

8.1 Chord: A distributed routing and location protocol

The Chord protocol solves the central problem: Given a key, locate the node responsible for that key. In this respect Chord can be considered a distributed hash table, where each node is a cell in the table (often called a bucket) that contains some keys.

The Chord system relies heavily on the use of identifiers consisting of m-bit unsigned integers ordered modulo 2^m, that is, in a circular identifier space. Each peer and each piece of data is associated with such an identifier, called the *key* and *node id*, respectively. In typical usage situations the following applies:

$$nodes \ll keys \ll 2^m$$

Chord takes advantage of the fact that both keys and node id's derive from the same identifier space in its strategy for assigning responsibility for a given key to a particular network node.

In the following we will describe how the problem is solved using consistent hashing, a simple strategy for mapping keys to nodes. This leads to a description of how the basic operations of the system are implemented and an analysis of the asymptotic runtime of these operations. Finally we summarise the most important characteristics of the Chord protocol and discuss what considerations must be taken into account due to the high degree of concurrency in the system.

8.1.1 Consistent hashing

The Chord protocol maps keys onto nodes in an m-bit circular identifier space, which is denoted the *Chord identifier circle* [21, 43]. The Chord protocol takes as input an m-bit key and returns the id of some node on the Chord identifier circle that holds the corresponding key. The Chord system uses a variant of *consistent hashing* [57, 58] to map keys to nodes.

Consistent hashing is a hashing technique which is designed to have the same advantages as standard hashing techniques, i.e. items are distributed evenly over a number of buckets and hash values are easily computed [54, 58]. Additionally consistent hashing addresses the problem of having a dynamic range of the hash function (a changing set of cells) [57]. In contrast to standard hashing techniques, different sets of cells do not induce completely different mappings of keys to cells when using consistent hashing. The mappings are instead "consistent" which means that for each different configuration of the hash table, the hash function does not completely reshuffle the mapping of keys to cells.

Consistent hashing can be implemented by mapping keys and nodes to points on a circle using a base hash function [58]. In Chord this is accomplished by assigning each node and each key a probabilistic unique m-bit identifier generated by a base hash function such as the cryptographic hash function SHA-1 [43, 44]. An identifier for a node is obtained by hashing a unique identifier of the node such as its IP-address in an Internet environment.

As mentioned above, Chord is based on a variation of consistent hashing. In consistent hashing, as in a conventional hash table, each bucket has global knowledge of all other buckets. In the Chord protocol the buckets (the Chord nodes) cannot know all other buckets in the system. The variation thus consists in the number of nodes that each node must be aware of in the system. Chord improves the scalability of consistent hashing by avoiding the requirement that every node must be aware of most other nodes in the system. [2] A Chord node needs only to be aware of a relatively small number of other nodes – this is described in section 8.1.4.

8.1.2 Mapping keys to nodes

A key is mapped to a node id in the following way: A key, k, is mapped to the first node whose identifier, id, is equal to or follows k in the identifier space (i.e. on the Chord identifier circle). The node responsible for k is defined as the *successor* of k's identifier [1]. This leads to the following definition of a successor:

Definition *The successor of an identifier, i, is the node with the smallest identifier which is greater than or equal to i.*

Figure 8.1 shows an example of a Chord identifier circle with a 3-bit identifier space and three nodes 2, 4 and 7. The three nodes are assigned a set of four keys, namely $\{$1,4,5,6$\}$. Successors are found in the clockwise direction on the identifier circle. Since node 2 is the successor of key 1, key 1 is assigned to node 2. Similarly, keys $\{$4,5,6$\}$ are assigned to their successor which is 4, 7 and 7 respectively.

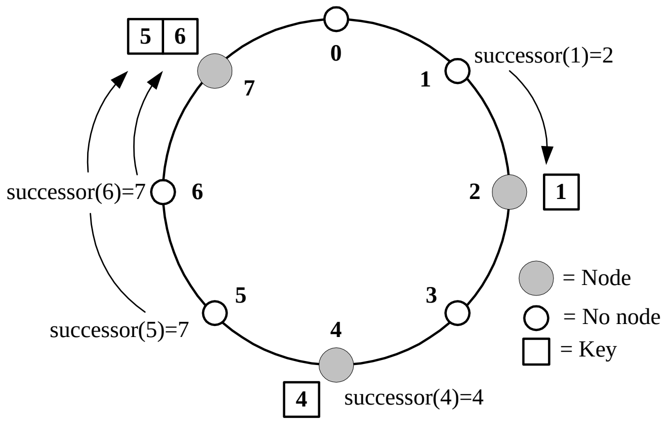


Figure 8.1: Mapping keys and associated values to nodes in a Chord identifier circle. Numbers (0-7) represent the identifier space. Keys $\{$1,4,5,6$\}$ are shown in little boxes next to the node which they are assigned. This figure is inspired by Stoica et al. [43].

If we assume that node 7 had not joined the Chord identifier circle yet, then the two keys 5 and 6 would have been assigned to node 2 since this node is the successor of the two keys. The successor of keys $\{$5,6$\}$ is found by wrapping around the identifier circle (see figure 8.2).

8.1.3 Operations in Chord

When describing the operations in Chord we distinguish between the Chord *protocol* and the Chord *system*. The protocol defines a single primitive function, `lookup`, that the operations in the system use. Given a key, the protocol looks up the node responsible for the key as shown in table 8.1.

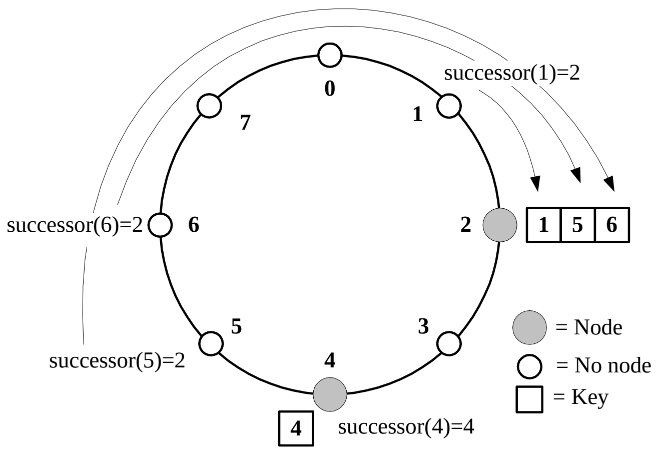


Figure 8.2: Illustration of how keys and associated values are assigned to nodes in a Chord identifier circle when wrap around occurs. Numbers (0-7) represent the identifier space. Keys $\{1,4,5,6\}$ are shown in little boxes next to the node to which they have been assigned.

On top of this simple operation more advanced operations can be built – this is called the Chord system. It is easy to implement a distributed hash table supporting storage and retrieval of $\langle$key,value$\rangle$ pairs when you can look up a node given a key. These operations[1] are shown in table 8.2.

Operation	**Description**
`lookup(key)`	Given a key, locate the node responsible for it.

Table 8.1: The central operation of the Chord protocol.

Operation	**Description**
`insert(key, value)`	Inserts a key and its associated value at the node found by `lookup(key)`.
`get(key)`	Returns the value associated with the key.

Table 8.2: The operations in the Chord system that maps keys to values.

Furthermore, the Chord system needs to allow nodes to enter and leave the network. For this purpose two methods are defined as shown in table 8.3.

In the following, we will describe the operations *lookup*, *insert*, *get*, *join* and *leave* in detail.

[1]In the Chord literature the operation that finds the node responsible for a value and the operation that loads the value given a key are both called `lookup(key)`. To avoid ambiguity we call the operation loading a value given a key `get(key)`.

Operation	**Description**
`join(node)`	Causes a node to add itself to the network via a node that already is in the network.
`leave()`	Causes a node to leave the Chord network.

Table 8.3: The operations in the Chord system that allow nodes to enter and leave the network.

Lookup A Chord node can perform a lookup in two ways: by using successor pointers or by using a *finger table*.

Chord uses only a small amount of routing information to implement consistent hashing in a distributed environment – information about successors suffices to perform a lookup. To look up the node responsible for a key, the `lookup` operation searches round the identifier circle via successor pointers until a node that succeeds the key has been reached – this is where the requested key can be found.

Figure 8.3 shows how a query of key 24 is passed on from successor to successor starting at node 3 and finishing at the immediate successor of 24 which is the node with identifier 26.

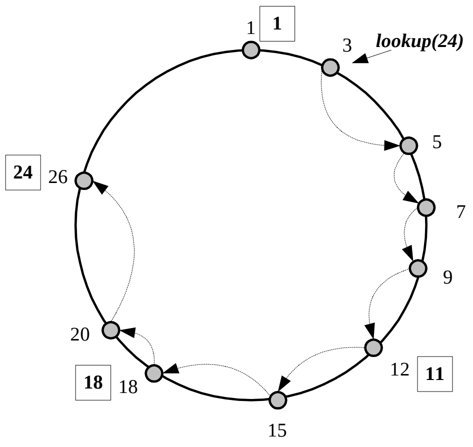


Figure 8.3: Illustration of how successors are used to perform a `lookup` operation (key 24 is queried). This Chord identifier circle has a 5-bit identifier space. Nodes are marked with gray dots (1,3,5,7,9,12,15,18,20,26). Keys $\{$1,11,18,24$\}$ are shown in little boxes next to the node which they have been assigned.

Performing lookups by stepping through the Chord identifier circle via successor pointers guarantees that the correct node is returned, but this strategy is inefficient – in the worst case every node needs to be traversed leading to a run time complexity of $O(N)$, where N is the total number of nodes in the system.

To improve the run time of `lookup` operations, additional routing information is needed: Every node maintains an m-entry table called the *finger* table, where

2^m is the size of identifier space. The finger table is maintained only to speed up `lookup` operations – the information is not essential for correctness as this can be achieved as long as the successor information is maintained correctly.

The i^{th} entry of the finger table of node n contains the identifier of the first node, s, which succeeds n by at least 2^{i-1} on the identifier circle [42]. That is:

$$s = successor(n + 2^{i-1})$$

The first entry in the finger table thus contains the immediate successor of n on the identifier circle. Note that the finger table is indexed from $1..N$, not $0..N-1$, so the immediate successor of a node is $finger[1]$. The idea of using fingers is to be able to move faster towards a key that has been queried, instead of simply traversing every node on the way. When using finger tables the distance to the key is approximately halved at each step until the node responsible for the key is reached [43].

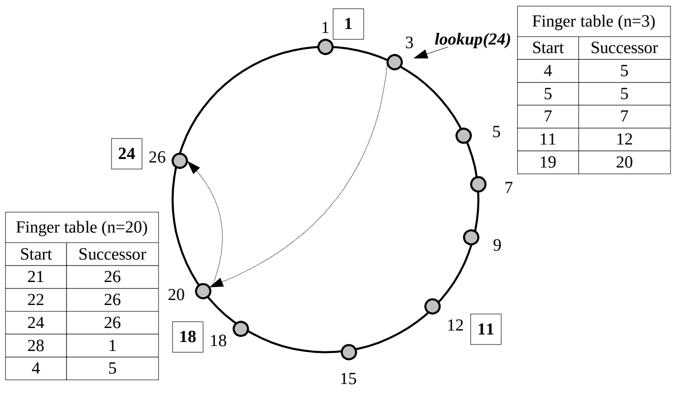
As a finger table does not contain information about successors of every identifier, a node may have to pass the request about the successor of a specific identifier, on to another node which is closer to the identifier. The request is passed on to the closest finger preceding the identifier.

Figure 8.4 shows how a lookup of key 24 is passed on using finger tables, starting at node 3 and finishing at the immediate successor of 24 which is 26. Only finger tables of node 3 and 20 are shown, as these are the only one used in this query.

The finger table of the node with identifier $id = 3$ stores information about the successors of the following identifiers:
$(3 + 2^0)\ mod\ 2^5 = 4$
$(3 + 2^1)\ mod\ 2^5 = 5$
$(3 + 2^2)\ mod\ 2^5 = 7$
$(3 + 2^3)\ mod\ 2^4 = 11$
$(3 + 2^4)\ mod\ 2^5 = 19$

The successor of identifiers 4 and 5 is 5, the successor of identifier 7 is 7, the successor of identifier 11 is 12 and the successor of identifier 19 is 20. Since node 3 does not hold any information about identifier 24, the query is passed on to the closest finger preceding 24 which is 20, found at entry 5.

Figure 8.5 shows the pseudo code for looking up a node responsible for a given key. The pseudo code uses set notation to denote whether an identifier is between two other identifiers (square brackets mean inclusive, rounded brackets mean exclusive). Note that the set operations are adapted to the circular identifier space. This means that the expressions $3 \in (1,1)$, $3 \in [1,1)$ and $3 \in (1,1]$ all are true – in Chord the node with id 3 *is* between node 1 and 1. The main loop of the algorithm is `findPredecessor()`, that finds the predecessor of the node responsible for a given key. Since `closestPrecedingFinger()` can never return a node greater than *id*, we will never accidentally overshoot the correct node. The process may undershoot though, but the check `id` $\notin$ `(n, n.successor]` in `findPredecessor()` ensures that we keep trying as long as we can find any successor closer to id. As long



Finger table (n=3)	
Start	Successor
4	5
5	5
7	7
11	12
19	20

Finger table (n=20)	
Start	Successor
21	26
22	26
24	26
28	1
4	5

Figure 8.4: Illustration of how finger tables are used to perform a `lookup` operation (key 24 is queried). This Chord identifier circle has a 5-bit identifier space. Nodes are marked with gray dots (1,3,5,7,9,12,15,18,20,26). Keys $\{1,11,18,24\}$ are shown in little boxes next to the node which they have been assigned. The finger tables of nodes 3 and 20 are shown.

as nodes maintain correct successor pointers, `findPredecessor()` will eventually succeed, regardless of whether the finger tables are incorrect.

```
lookup(key)
    n = findPredecessor(key);
    return n.successor;
findPredecessor(id)
    n = this;
    while (id ∉ (n, n.successor])
        //n = n.successor; // Slow lookups
        n = n.closestPrecedingFinger(id); // Fast lookups
    return n;
closestPrecedingFinger(id)
    for i = M downto 1
        if (finger[i].node ∈ (this, id))
            return finger[i].node;
```

Figure 8.5: Pseudo code for the lookup operation

Insert and get These operations make it possible to insert a ⟨key,value⟩ mapping at a node, and to retrieve a value given a key, respectively. Both simply consists of looking up the node responsible for storing the given key and then executing either a `load` or `save` operation on the node.

Join and leave In a dynamic environment nodes can join and leave the Chord identifier circle at any time.

To simplify the `join` and `leave` operations each node maintains a *predecessor* pointer, a pointer to the node that immediately precedes it. The predecessor pointer can be used to traverse the nodes on the identifier circle in a counterclockwise direction.

Whenever a node is joining or leaving the identifier circle this change has to be reflected in finger tables, predecessor pointers and in assignment of keys to nodes.

If a new node, n, joins the identifier circle, n may have to be added to the finger table of other nodes on the identifier circle. Therefore, it is necessary to update finger tables of other nodes on the identifier circle to reflect the joining of n. It is also necessary to reassign every key whose successor has become the new node n.

Consistent hashing ensures that a minimal number of keys have to be moved when nodes join and leave the network. To maintain the consistent mapping when a node joins or leaves the Chord identifier circle (network), keys may have to be moved. If a node n joins the identifier circle, keys assigned to the successor of n have to be checked to see if they should still be assigned to the successor of n, or if they should be moved to the new node n. If a node n leaves the identifier circle, all of the keys assigned to n are reassigned to the successor of n. Figure 8.6 illustrates how keys are reassigned when a node (with identifier 6) joins the identifier circle (figure 8.6, b) or a node (with identifier 2) leaves the identifier circle(figure 8.6, c). Since joining and leaving the Chord identifier circle only introduces movement of keys between the joining/leaving node and its successor, a complete reshuffling of keys is not necessary – only an $O(1/N)$ fraction of keys are moved to a different location, where N is the total number of nodes in the Chord system. This means that most keys are mapped to the same node as they were before the change.

8.1.4 Properties of Chord operations

In this section the properties of the Chord operations `lookup`, `join` and `leave` are described and analysed. Note that all complexity estimates are probabilistic (expected values) and not upper bounds.

Lookup

In an N node Chord system, each node maintains information about $O(\log N)$ other nodes. The information is, as mentioned above, stored in a finger table, and is used to improve the runtime of `lookup` operations. Using the finger tables to perform a lookup this operation requires $O(\log N)$ messages [43]. If instead a lookup is performed using successors, the operation requires $O(N)$ messages.

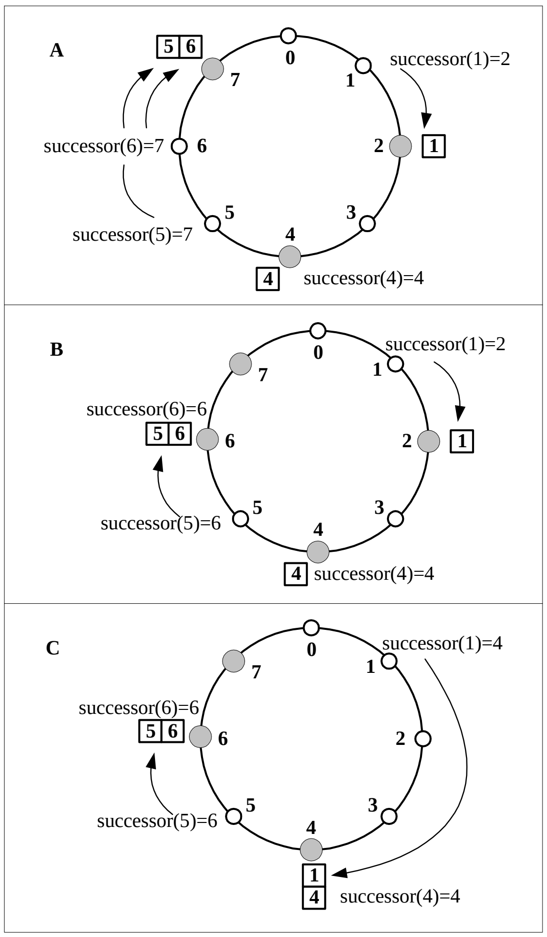


Figure 8.6: Illustration of how keys and associated values are reassigned to nodes in a Chord identifier circle when a node joins (B) or leaves (C) the circle. A represents the original Chord identifier circle. B represents the Chord identifier circle after a node with identifier 6 joins the circle. C represents the identifier circle after the node with identifier 2 leaves the circle. Numbers (0-7) represent the identifier space. Nodes are marked with gray dots (2,4,6,7). Keys $\{$1,4,5,6$\}$ are shown in little boxes next to the node to which they have been assigned.

Join

Three tasks must be performed when a node n joins the Chord network:

1. Initialise predecessor and finger table of n (including successor information).
2. Update predecessor pointer and finger tables (including successor information) of other nodes to reflect the joining of n.
3. Transfer keys for which n is now the successor, from the successor of n to n.

Complexity of step 1 Finding and initialising the predecessor of the new node n takes $O(\log N)$ messages, where N is the number of nodes on the Chord identifier circle. It takes $O(\log N)$ to initialise each entry of the finger table of n (each entry is obtained from a lookup operation), and there are m entries in the finger table. This yields a total of $O(m \log N)$, where m is the number of bits required to represent the identifier space and N is the number of nodes on the Chord identifier circle.

Stoica et al. [42] have shown that the complexity can be reduced to $O(\log^2 N)$, if n checks whether the i^{th} finger is also the correct $(i+1)^{th}$ finger for each i.

Complexity of step 2 The predecessor pointer of only one node needs to be updated. We need to update *one* entry of $\log N$ nodes (proved by Stoica et al. in [44]), and each update takes time $O(\log N)$. This gives us a complexity of $O(\log^2 N)$.

Complexity of step 3 Stoica et al. [43] has shown that each node in the Chord identifier circle is responsible for at most $(1+\epsilon)K/N$ keys, where K is the size of the set of keys. Lewin [58] has shown that ϵ can be minimised to a small constant, if each node runs $O(\log N)$ *virtual nodes*, each with its own identifier.

When a node joins the Chord identifier circle, $O(K/N)$ keys have to be moved.

Total complexity of join The above three steps yields a total complexity [2] of:

$$O(\log^2 N) + O(\log^2 N) + O(K/N) = O(\log^2 N + K/N)$$

[2]The Chord articles claim a complexity of $O(\log^2 N)$, but they only consider how many messages are needed for re-establishing the routing invariants i.e. each node's finger table is correctly filled and each key k is assigned to node *successor*(k), whereas we also take the cost of moving keys into account.

Leave

The analysis of the complexity of the `leave` operation is similar to the above analysis of the complexity as a `join` operation and yields the same complexity as `join`. However, the first step, which initialises the predecessor and finger table of the joining node, should of course be omitted. The total complexity of `leave` is therefore equal to the complexity of `join`, namely $O(\log^2 N + K/N)$.

8.1.5 Concurrency

Due to its distributed nature, the Chord protocol needs to be able to handle massive concurrency as all peers in the system can act simultaneously. Managing concurrency is especially important when nodes join and leave the system. Concurrent joins and leaves from the Chord identifier circle are handled by maintaining the invariant that every node is aware of its immediate successor ($finger[1]$) and predecessor, and by allowing the remaining entries of every node's finger table to converge to a stable state over time [21].

If two or more nodes with almost similar identifiers are joining the Chord identifier circle at the same time, inconsistencies can occur since more than one node may try to notify the same predecessor that they are its new immediate successor. The method for setting the predecessor ensures that only the node with the lowest identifier will succeed in notifying the predecessor of itself. The remainder of the joining nodes will learn their correct immediate predecessor and successor by periodic calls to the `stabilise` operation. The `stabilise` operation periodically checks whether new nodes have inserted themselves between a node and its immediate predecessor and successor, when joining the Chord identifier circle [42, 44]. When a node n runs the `stabilise()` method, it asks its successor s for s's predecessor p, and decides whether p should be n's successor instead. This could be the case if p had joined the identifier circle very recently. An operation similar to `stabilise` periodically updates entries of each node's finger table, which causes the tables to gradually converge on correct values after a join [44]. Pseudo code for the operations is shown in figure 8.7.

8.2 Caching

This section presents the caching strategy used in the CFS system and in XML Store system, and discusses the advantage of working value-oriented when using caching. Caching is a standard technique for storing data temporarily to improve performance of later retrieval. A cache is a small fast storage area holding recently accessed data, which is designed to speed up subsequent access to the same data [59]. For instance, caches can be used to hold local copies of data which is accessible over a network. Thereby the need for remote access to data is diminished.

```
stabilise()
    x = predecessor.successor;
    if ( x ∈ (predecessor, this) )
        predecessor = x;
    x = successor.predecessor;
    if ( x ∈ (this, successor) )
        successor = x;
fixFingers()
    i = 1 < random index <= M
    finger[i].node = findSuccessor(finger[i].start);
```

Figure 8.7: Operations for stabilising the network after concurrent join- and leave-operations.

Caching strategy

Our caching strategy has been adopted from the CFS system [1, 2]. When a value is to be retrieved from the XML Store system using a lookup operation, the local cache is checked to see if the required value is already in the cache. If the value is in the cache (a cache *hit*) then it is returned immediately and the lookup terminates. However, if the value is not cached (a cache *miss*) then the lookup operation starts searching the Chord identifier circle until the requested value has been found. Each time a server is encountered on the lookup path, the cache of the server is checked for a cache hit. When the value is found and returned, a copy of the value is saved in the cache of each of the servers on the lookup path.

LRU caching

As a cache is of a limited size, some sort of mechanism is needed to ensure that the cache contains "relevant" data. We have chosen to use the same cache replacement technique as used in the CFS system, namely LRU caching [1, 2]. LRU (*Least-recently-used*) caching is one of the most widespread and important replacement algorithms developed for main memory and disk caching [60]. LRU caching works in the following way: Recently used data is stored locally in a cache. The cache is ordered from the most recently used to the least recently used piece of data. As the cache is filled up with recently used data, the "oldest" cache entries (i.e. the data which has not been used for the longest time) are discarded, to stay below the maximum limit [61].

Cache consistency

When working with updateable data, consistency problems might occur, since all cached copies need to be kept up to date [5]. If the cached copies are not updated when the original data changes, the value of the cache is greatly reduced. A variety of cache consistency protocols have been developed to ensure that cached copies eventually will reflect changes to the original data, including time-to-live, time stamping, client polling and invalidation protocols [5, 62].

As described in chapter 3, a characteristic of the value-oriented paradigm is that values are immutable and therefore cannot be updated. Cache consistency problems are consequently avoided when working value-oriented.

8.3 Load balancing

Load balancing refers to the problem of spreading data evenly among peers, according to their capacity.

In this section we briefly describe how load balancing is handled in the CFS system, namely partly by means of consistent hashing and partly by means of *virtual servers* [1, 2].

Load balancing and consistent hashing

Chord uses consistent hashing to map values to nodes (see section 8.1.1), and thereby values are spread evenly around the identifier space (the Chord identifier circle). Files are split up in both the CFS system and the XML Store system, and thereby file content is spread over many peers. This means that requests for popular files will affect several peers instead of just a single peer, which provides some degree of load balancing. It is however, not nearly enough to produce perfect load balance. Due to the uniform distribution of values every peer in the system stores roughly the same quantity of data, no matter how much storage capacity each peer actually has. Furthermore, if the identifier space is large then peers will most likely be placed at the Chord identifier circle with irregular spacing. This will cause some peers to get responsibility of a larger amount of data than other peers, even though data is “uniformly” distributed with consistent hashing. Figure 8.8 illustrates a Chord identifier circle with nodes of irregular spacing. All keys between X and Y will be placed at node Y.

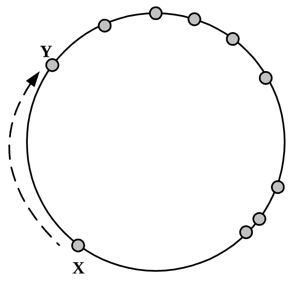


Figure 8.8: Illustration of a Chord identifier circle with nodes of irregular spacing. All keys between X and Y will be placed at node Y. Gray dots represent nodes.

Virtual servers

The CFS system seeks to improve load balancing by taking heterogenous peer capacities and irregular spacing between peers into account. This is accomplished by using so called *virtual servers* [1, 2]. A single server is configured so that it will act as multiple virtual servers, thereby taking responsibility for a number of identifiers and not just one. Introducing virtual servers will increase the total number of peers on the Chord identifier circle, and the problem of irregular spacing will consequently be reduced. Each “real” server is configured with a number of virtual servers corresponding roughly to the peer’s storage capacity, taking heterogenous capacities into account. A virtual server is identified by the IP address of the real server combined with the index of the virtual server within the real server.

When introducing virtual servers to the system the number of peers on the Chord identifier circle is as mentioned increased. To prevent increasing the number of hops in a lookup operation proportionally, virtual servers on the same physical machine are allowed to examine each other’s finger tables.

The use of virtual servers however, reduces the advantages of replication to some extent, as failure of peers are no longer independent. If one physical machine fails a large group of peers fail simultaneously.

Caching

Together with the use of virtual servers and by spreading blocks evenly across the network, caching takes part in improving load balancing. By means of caching of data at an appropriate number of peers, overloading of peers holding popular data can be further reduced.

8.4 Fault tolerance

In this section we discuss how fault tolerance is handled in the CFS system. Dabek et al. [1] seek to increase availability and robustness of the CFS system by means of replication and *successor lists*. Fault tolerance is not implemented in the XML Store, but all the described techniques can be applied without modification.

Successor list

To increase robustness, each node in the Chord system maintains a successor list of size r. The successor list contains the first r successors of a node. If the immediate successor of a node does not respond, the node can use the next “successor” in its successor list, to be able to perform a lookup. This way the probability of disrupting the Chord identifier circle is decreased. Actually all r successors would have to simultaneously fail to disrupt the identifier circle. Dabek et al. [1] suggest a successor list of length $O(\log N)$, where N is the number of nodes on the Chord identifier circle.

Replication

To improve availability of data, the CFS system makes use of replication. The replication scheme from the CFS system [1, 2] is presented below.

Each block of data is replicated on k different servers. The replicas are placed at the k servers that immediately follow the server where the original piece of data is stored (i.e. the successor of the id of the data on the Chord identifier circle). The placement of replicas means that if a server s crashes, the data which is stored on s is immediately available at the successor server (ss) of s. The successor server (ss) is able to determine whether it should take responsibility of the data through examination of its finger table; if the identifier of the data is between the predecessor of ss and ss, then ss should take responsibility of the data. This means for instance, that ss is now responsible for making replicas of the data.

Since server ids are obtained by hashing the IP address (and possibly an index if virtual servers are used), servers which are physically close to each other are not likely to be close to each other on the Chord identifier circle. Therefore data will most likely survive local network failures due to the replication of data to successor servers on the Chord identifier circle. This provides an independence of failure to the CFS system.

The need for multiple replicas of files significantly increases the storage demand. However, since disk space becomes more and more inexpensive, replication is considered acceptable when wishing to achieve availability and robustness.

Replication and successor lists make sure that a substantial number of nodes have to crash to make the system lose data or make the routing fail.

8.5 Server selection

In the following section the implementation of server selection in CFS is explained and discussed. Server selection has been added to CFS to reduce lookup latency by allowing lookups to preferentially contact peers likely to be nearby in the underlying network [1, 2]. Server selection has not been implemented in the present XML Store prototype, but can be applied without modification.

Server selection in CFS

To be able to determine which peer is closest, latency estimates are kept at each peer in CFS. Latencies are measured while building finger tables [1, 2].

When performing a `findPredecessor(id)` operation the next peer to hop to is chosen from a set of peers, namely the finger table of the current peer. When choosing a peer from the set, the choice is based on the latency estimates found at that peer [1, 2]. Different choices of peers to hop to, will cause the lookup query to go different distances around the Chord identifier circle.

Problems regarding server selection

Estimates for latencies which are accurate and at all time up to date would certainly reduce lookup latency. However, unreliable latency estimates could do more harm than good by possibly increasing lookup latency. The server selection strategy described in the CFS system [1, 2] might disturb the $O(\log N)$ run time complexity of lookup operations. This is due to the fact that choosing a peer from the finger table based on latency instead of proximity on the Chord identifier circle can potentially result in a linear run time, $O(N)$, if the chosen peer is always the immediate successor. Experimental results have shown that download times are at least modestly improved when applying server selection [2].

8.6 Summary

In this chapter we have shown that it is possible to build a distributed peer-to-peer storage area for storing $\langle$key,value$\rangle$ pairs using the Chord protocol. This fits the requirements for a storage layer for the XML Store system very well. The Chord protocol has two central properties that make it extremely suited for building a distributed storage area. First of all it is remarkably scalable: Lookup operations can be achieved using $O(\log N)$ messages, where N is the number of peers in the system. This means that lookup operations are feasible even in very large systems [1, 43]. Secondly, Chord is fully distributed – no node holds more information than any other node in the system. This decentralisation makes Chord suitable for peer-to-peer applications – there is no single vulnerable point.

The basic Chord protocol can be extended with features that can improve the systems performance or increase its stability:

- Improved load balancing using *virtual servers*
- *Server selection* to approximate network distance and select the nearest peer
- Caching for faster access to data
- Fault tolerance by maintaining a list of successors
- Availability of data by replication

These features are required to make the system fast and stable in a realistic setting. It has been shown that it is relatively easy to add techniques for failure handling and stability to systems built using the Chord protocol without crucial loss of performance [1, 2].

We have chosen not to focus on failure handling and stability, but to concentrate on the scalability and decentralisation properties of the system. We will not implement virtual servers, replication and successor lists. We will implement caching.

Chapter 9

Value-oriented storage

This chapter discusses the value-oriented storage strategy. We start with a brief description of the interaction between the *XML Storage* layer and the *Distributed Storage* layer, since this interaction is essential to the storage strategy. Next, value references and the storage strategy are discussed and finally we analyse different implementation strategies of the *Disk* layer.

9.1 XML Storage and Distributed Storage layers

In this section we discuss the interaction between the *XML Storage* layer and the *Distributed Storage* layer. Recall that the XML Storage layer in cooperation with the XML layer offers the application programmer a tree structured object representation of XML documents and allows storage and retrieval of these documents. The Distributed Storage layer consist of a peer-to-peer network based on the Chord protocol. It works with primitive data-types, offering storage and retrieval of byte-sequences associated with keys consisting of 160-bit unsigned integers.

The main idea is to use the Distributed Storage layer as a service that translates a location independent value reference to a network route where the value can be found. The Distributed Storage layer acts as a value-oriented hash table that stores values (sequences of bytes) associated with value references (keys). Combining the Distributed Storage layer with the concept of a value reference, makes it possible to find a certain value regardless of where the value is actually stored. In this way information is divorced from location.

The Distributed Storage layer is inspired by CFS [2], and even though CFS was not conceived as a value-oriented file system, but rather a *read-only* file system, it does have some value-oriented traits in that the stored blocks of data cannot be updated or deleted. The Distributed Storage layer has potential to act as a value-oriented storage layer in many contexts, as the definition of a value (a sequence of bytes) arguably is the most general possible. This permits all kinds of data to be stored in value-oriented way, not just XML documents.

9.2 Value references

In this section we describe how value references are implemented as 160-bit unsigned integers using cryptographic hashing, making them compatible with the keys used by the Distributed Storage layer and the Chord protocol. Value references play an important role in the XML Store system, so we will look into the implementation of these in some detail.

In chapter 3 the concept of a value reference was introduced. Recall, that a value reference is an identifier of an immutable value, with a set of desired properties outlined in section 3.1.

To obtain a value reference of a value consisting of a finite sequence of bytes, we use a cryptographic hash function (SHA-1). The basic idea of cryptographic hash functions is that a hash-code serves as a compact representative image (sometimes called an imprint, digital fingerprint, or message digest) of an input string, and can be used as if it were uniquely identifiable with that string [63]. SHA-1 is a one-way hash function that takes arbitrary-sized data and outputs a fixed-length hash value. A value reference in our system is simply a cryptographic hash value obtained by hashing the byte representation of the value. Implementing value references in this way was suggested by Henglein [9], and the approach is also similar to the way CFS identifies blocks of data [2], although they do not use the term value reference.

By implementing value references in the described way, we achieve two of the desired properties, namely that the value reference is a deterministic function of the value alone and that the value reference has a limited size. The third desired property (a value reference is injective) is impossible to achieve using cryptographic hashing, as a key has a fixed size and the number of byte-sequences of varying size is infinite. However, due to the size and nature of the key, a value reference is unique with very high probability ($1 - 2^{-160}$).

9.2.1 Collisions

Even though a collision is highly unlikely when using a cryptographic hash function, a collision could nevertheless be catastrophic. It would either lead to critical data not being stored or existing data being “overwritten”, as data items would be mistaken for each other due to the identical value reference. There are two solutions to this problem: The first is to ignore the problem because a collision is far too unlikely to occur. The second is to devise a scheme for detecting collisions. In the following section we will elaborate on the probability of a collision and describe a protocol that detects collisions before they occur.

Probability of collisions

Finding out how many different values it takes before the chance of a collision reaches a certain level is identical to the statistical paradox, known as the *birthday* *paradox*: The probability of finding a matching pair in a given set is far greater than for finding a match for a given individual [5]. The formula (using Stirlings approximation) for calculating the number of individuals required to reach a given probability of a match is:

$$\sqrt{\frac{\pi \times M}{prob}}$$

M is the identifier space (in our case 2^{160}) and $prob$ is the inverse probability of a collision. For instance, $prob = 2$ means that there is a 50% chance of a collision, whereas $prob = 1000$ means that there is a 0.1% chance of a collision. If we for example choose to accept a 0.1% risk of collision, we can store 6.776×10^{22} different values. In the present XML Store prototype we simply choose to ignore collisions.

Collision detection

Clarke et al. [17] have described a simple technique for detecting collisions. Consider the following scenario: We wish to save a value v with key (value reference) k. Before saving v it is checked whether k is already saved in the system. If k *does not* exist in the system, v is stored and associated with k in the normal way. If k, on the other hand, *does* exist in the system, the previous value associated with k is retrieved and compared to v. If the pre-existing value is equal to v then everything is fine, and the save operation can be considered accomplished – the value is already in the system and there is no need for saving it again. If a collision has occurred – that is the pre-existing value differs from v – some action must be taken. The obvious solution would be to change the value slightly by adding some recognisable form of padding. Changing the value results in a different key, as the key is a content hash of the value. The save operation can now be repeated for a new $\langle$key,value$\rangle$ pair and will have just as high a chance of succeeding as the first save operation.

As frequent equality testing of large values could prove expensive, optimisations for determining whether the pre-existing value is equal to v, can be considered. Instead of comparing the values byte by byte, the length (in number of bytes) and a prefix consisting of the first n bytes of the values could be compared. If the lengths and the prefixes of both values are equal there is a good chance that the values are indeed equal. This is due to the fact that a cryptographic hash function often produces very different keys from similar values.

9.2.2 Preventing forging

Using encrypted hash codes allows us to detect forging. When a client requests the value corresponding to a certain value reference, she can check whether the actually received value is consistent with the value reference.

9.3 Storage strategy

The XML Storage layer is responsible for storage and retrieval of XML documents represented as a node structure. In the following we explain how the XML Storage layer employs the Distributed Storage layer for storage of the XML documents. The Distributed Storage layer is based on the Chord protocol and offers the following simple interface that allows the storage and retrieval of byte-sequences associated with an identifier.

```
void save(id, byte[])
byte[] load(id)
```

As previously mentioned, this interface can easily be translated to store $\langle$value reference,value$\rangle$ pairs, with a value reference acting as a Chord identifier associated with a value.

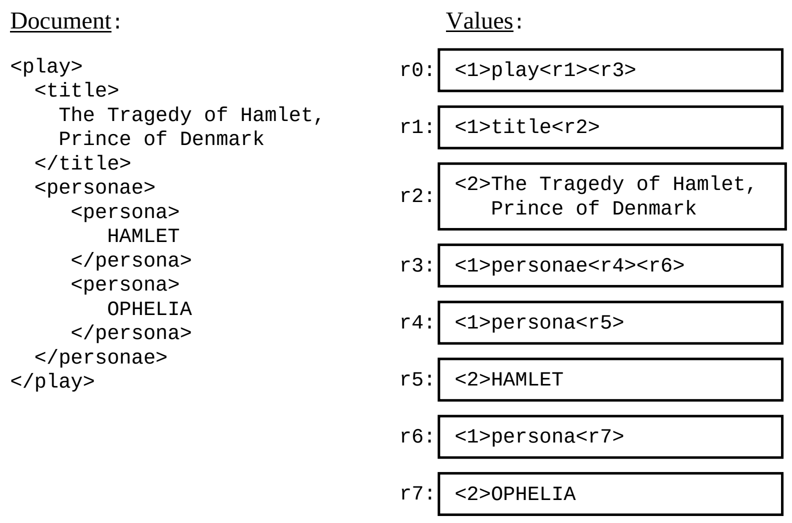
The strategy for transforming an XML document to one or more byte chunks is important as the format chosen has severe consequences for the system. It is of course possible to store an XML document in the usual serialised text format, but not much would be gained compared to existing technologies such as DOM and SAX. The document would only be serially accessible as it would have to be read in its entirety every time it was loaded into the object structure. Identical parts of the document would not be shared as it would be impossible to identify the tree structure in a single flat file. In addition, it would be a poor use of the Chord protocol, that requires files to be split up to help achieve load balancing.

Instead we propose a storage strategy that stores each node as a separate value. When an XML document is saved the document's tree structure is traversed and each node is saved as a separate value. A value reference is generated by hashing the byte representation of the node and the $\langle$value reference,byte-representation$\rangle$ pair is placed on the appropriate Chord server located by the Distributed Storage layer using the Chord protocol.

For simplicity we use a string based format for representing the nodes as bytes, even though a binary format might be more efficient. The representation of a node always begins with a flag indicating the type of the node (`<1>` for elements, `<2>` for character data). A character data node is stored by simply serialising the text it contains. An XML element is stored as a string containing the name of the element and the value references to its children. Figure 9.1 illustrates the storage strategy by showing an XML document and the values that it consists of when stored. Note that the value references have been simplified to increase readability – they are normally numbers between 0 and $(2^{160} - 1)$.

9.3.1 Sharing

An important advantage of the suggested storage strategy is that it makes sharing of identical parts between documents possible. When a document is loaded and part of it is modified we only have to save the parts of the document affected by



Document:

```
<play>
  <title>
    The Tragedy of Hamlet,
    Prince of Denmark
  </title>
  <personae>
     <persona>
        HAMLET
     </persona>
     <persona>
        OPHELIA
     </persona>
  </personae>
</play>
```

Values:

```
r0: <1>play<r1><r3>
r1: <1>title<r2>
r2: <2>The Tragedy of Hamlet,
       Prince of Denmark
r3: <1>personae<r4><r6>
r4: <1>persona<r5>
r5: <2>HAMLET
r6: <1>persona<r7>
r7: <2>OPHELIA
```

Figure 9.1: Illustration of an XML document and its values when stored in the XML Store system.

the modification, namely the path from the root to the modification (as described in chapter 3). The remaining nodes are already stored and there is no need for storing them again. This optimisation can be implemented by simply adding a flag to each node indicating whether or not the node is already stored. The flag is set when the node is stored for the first time or when the node is loaded from the XML Store, indicating that it is indeed already on disk. If one for example wished to add information about the playwright of the play to the document in figure 9.1 by adding `<author>Shakespeare</author>` to the root element `<play>`, then both the `title` and `personae` subtrees would remain unchanged and would not need to be stored again. This could potentially save a lot of storage space under the assumption that most XML documents are copies or slight modifications of other XML documents.

9.3.2 Lazy loading

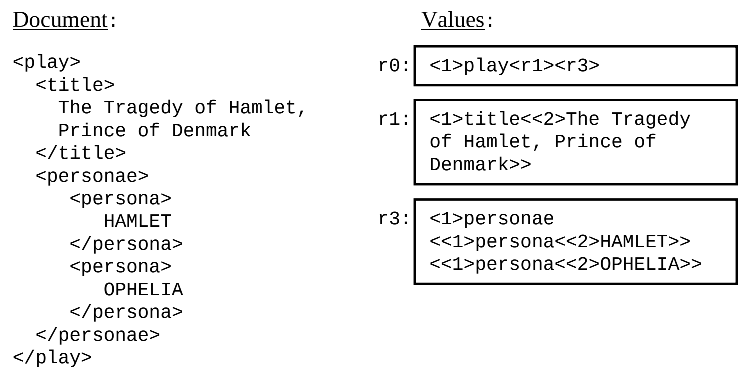
Another important advantage of the storage strategy is that it allows the application programmer to traverse and access parts of a document without loading the entire document into memory. The title of the play in figure 9.1 can for example be accessed by traversing the `play/title` path, making it unnecessary to load the entire play including possibly a very large number of lines and scenic descriptions. The node structure hides this from the application programmer by employing lazy loading. This is accomplished by letting the child nodes of an element be proxy nodes that only know their own value reference. When asked for content the proxy simply loads the actual node.

These two properties combined makes the storage strategy very efficient when modifying, storing and retrieving large XML documents.

9.3.3 Evaluation of storage strategy

The strategy of storing each node separately is highly efficient for sharing purposes – identical parts of documents are guaranteed to be shared at node granularity. The strategy does unfortunately not perform very well as a document is split up into many small parts that are saved independently. Each separate save operation is very expensive, due to the network overhead associated with it.

To prevent the document from being split up into too many small parts we suggest an alternate storage strategy, where several nodes are grouped in a single block of data. Instead of including a reference to the subtree, the byte representation of the subtree is simply inlined. The tree is traversed in post-order and only when the subtrees have exceeded a certain limit they are saved. Figure 9.2 shows an XML document and the values that it consists of when applying this technique.


Document:

```
<play>
  <title>
    The Tragedy of Hamlet,
    Prince of Denmark
  </title>
  <personae>
     <persona>
        HAMLET
     </persona>
     <persona>
        OPHELIA
     </persona>
  </personae>
</play>
```

Values:

```
r0: <1>play<r1><r3>

r1: <1>title<<2>The Tragedy
    of Hamlet, Prince of
    Denmark>>

r3: <1>personae
    <<1>persona<<2>HAMLET>>
    <<1>persona<<2>OPHELIA>>
```

Figure 9.2: The alternate storage strategy. Nodes are grouped and a certain value-size is used.

We accomplish our main goal, to decrease the number of save operations required, by generating fewer and larger pieces of data. We still maintain a tree structure on disk, but with fewer and larger nodes. This means that we still only have to change nodes from the place where the modification took place to the root of the tree. Identical subtrees are still shared, but the amount of sharing is reduced, because identical nodes that previously would have been shared now can be inlined in different contexts. The proposed alternate storage strategy has not been implemented in the present XML Store system. Therefore it is hard to tell whether the strategy has a serious impact on sharing.

9.4 Disk handling

This section analyses different implementation strategies of the *Disk* layer of the XML Store system.

The Disk layer of the XML Store must be able to persist ⟨byte sequence,value reference⟩ pairs to the local file system for permanent storage. The layer must also support deletion of values, since the Distributed Storage layer requires that values are moved around when other nodes take responsibility for a value. This leads to the following simple interface:

Operation	**Description**
`void save(byte[], ref)`	Saves a sequence of bytes associated with the value reference *ref*.
`byte[] load(ref)`	Loads a value with the value reference *ref* and returns the value as a sequence of bytes.
`boolean delete(ref)`	Deletes a value given its value reference *ref*. Returns a boolean indicating if the deletion was successful.
`boolean contains(ref)`	Returns true if the disk contains the value with value reference *ref*.

Figure 9.3: The `disk` interface of the XML Store.

9.4.1 Each value in a separate file

One way to implement the above interface is to store each data item of data in a separate file. This strategy is simple to implement, but also very inefficient, since many small files cause a large overhead regarding disk space. The minimum file size on conventional file systems is typically about 4 kB, which means that a value that contains only a few bytes of data takes up 4 kB of space on the hard disk. This strategy is also inefficient time-wise, since the calls to the operating system needed for creating new files are very expensive.

Given a value reference the disk layer needs a location resolver to locate the corresponding file. The location resolver can be implemented as a hash table that maps value references to file names. When a value is saved, the value reference of the value and the name of the created file are inserted into the location resolver, and when the value later needs to be loaded, the file name is looked up in the location resolver using the value reference of the requested value.

The `delete()` operation can be implemented by deleting the actual file and removing the corresponding entry in the location resolver table.

9.4.2 Log-structured disk

An alternative to the aforementioned solution is an implementation that is inspired by a log-structured file system. A log-structured file system does not support saving a value at a particular place on disk. Instead the system allocates all values in a sequence in, like in a log [64]. When a value is saved, the system returns a locator for the value, which can then be used for future retrieval of the value. A log-structured file system can be simulated on a conventional file system such as the NT File System (NTFS) by using large random access files, each containing multiple blocks of data. A locator can be implemented as a pair of integers. The first integer is an offset to the file position where the value is located, and the second tells the length of the data in bytes. The relationship between locators and a random access file is illustrated in figure 9.4.

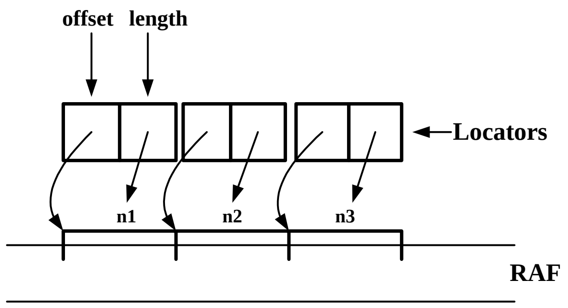


Figure 9.4: Illustration of how locators are used to locate a file in a random access file (RAF). n1-n3 illustrates arbitrary sized blocks of data.

In this solution the location resolver is a hash table that maps value references to locators.

The `delete` operation can be implemented by adding a flag to each entry in the location resolver indicating whether or not the value is in use. The unused values can then be removed by a "cleaner", that is periodically invoked. The cleaner can be implemented by using the *copying garbage collection* scheme [65].

9.4.3 Our choice of implementation

The implementation inspired by the log-structured file system is obviously the most efficient of the two sketched solutions, regarding both space and time consumption. By using this design it is possible to achieve a space consumption competitive to the size of the typical flat text file format of XML documents, even though we store each XML document as many small blocks of data. This has been done by Pedersen et al. [66]. However, since we do not focus on efficient disk handling in our implementation of the proof-of-concept prototype we choose to save each value in a separate file.

Chapter 10

Symbolic names

To use the XML Store to build useful distributed applications, we need to be able to associate XML documents with names readable by humans rather than 160-bit numerical identifiers. Clients cannot share particular resources managed by a computer system unless they can name them consistently. Thus, names facilitate communication and resource sharing [5].

It should be possible to update the association between a symbolic name and a document since the entire world is not value oriented: Exchange rates change, newspapers publish new articles, the weather forecast changes etc. Consider for instance a weather forecast service: Clients wishing to receive the most recent weather forecast need to have a way of locating the document representing the current forecast. Having a shared, updateable name, such as `xmls://dmi.dk/forecast`, makes this possible. Consequently, we need to augment the XML Store with a *Name Service*, that allows us to associate human readable names with XML documents stored in the XML Store.

In this section we initially define the concepts of a *name* and a *name service*. We then describe the Directory, a simple name service used by the XML Store system. Updateable references shared by multiple clients introduce concurrency problems. We describe these problems and some possible solutions for handling transactions. Finally, we suggest how the Chord protocol could be used for implementing a more advanced distributed name service.

10.1 Names and name service

A name is a (preferably human-readable) string that belongs to a given *name space*. A name space is the collection of all valid names recognised by a particular name service. The association between a name and a resource is called a *binding*. A name service stores a collection of one or more naming contexts – sets of bindings between names and resources. The most central operation a name service supports is to *resolve* a name, that is to look up a resource given a name. [5]

You could argue that value references, in a sense, are names for values. However, value references are not adequate for our purpose for two reasons: First of all, they are *identifiers*, that is, names whose primary purpose is for interpretation by a computer, and thus not particularly human-readable. Most importantly, though, they cannot be updated to refer to a different value, as they have an injective relation to the value that they identify.

Names can be *pure*, in that they are uninterpreted strings, containing no information about the location of the resource. *Non-pure* names contain some degree of information about the location of the object they name. Names consisting entirely of location information are called *addresses*. A Uniform Resource Locator (URL), as known from the World Wide Web, is an address: It specifies the location of a particular web page on a specific web server [1]. Addresses are efficient for accessing resources, but as resources are often relocated, addresses are inadequate for identification. [5]

10.2 The XML Store Directory

In the XML Store system we propose a simple name service, called the *Directory*, that maps names to value references. We have devised the name service so it is completely separated from the XML Store. The XML Store can function in its own right, without the Directory. The Directory operations are summarised in table 10.1. The `update()`-operation will be described in more detail in section 10.3, as it deals with concurrency problems when performing updates.

Operation	**Description**
`void bind(name, ref) throws NameAllreadyBoundException`	Binds a name $name$ to a value reference ref
`ValueReference unbind(name) throws NoSuchElementException`	Removes the binding of a name $name$
`ValueReference lookup(name) throws NoSuchElementException`	Resolves the value reference bound to a name $name$
`void update(name, new, expected) throws ConcurrentUpdateException, NoSuchElementException`	Updates the binding of a name $name$ from a value reference $expected$ to a new value reference new

Table 10.1: The operations supported by the XML Store directory.

To retrieve a given XML document from the XML Store we only need the value reference of the document. The Chord protocol is responsible for locating the document in the Distributed Storage layer. By letting the Directory map names to value references we obtain a pure name space, where the name of a document

[1]A URL can, however, also be viewed as a non-pure name, since the host name can be resolved into an IP-address by the DNS.

is completely independent from the location of the document. This is illustrated in figure 10.1.

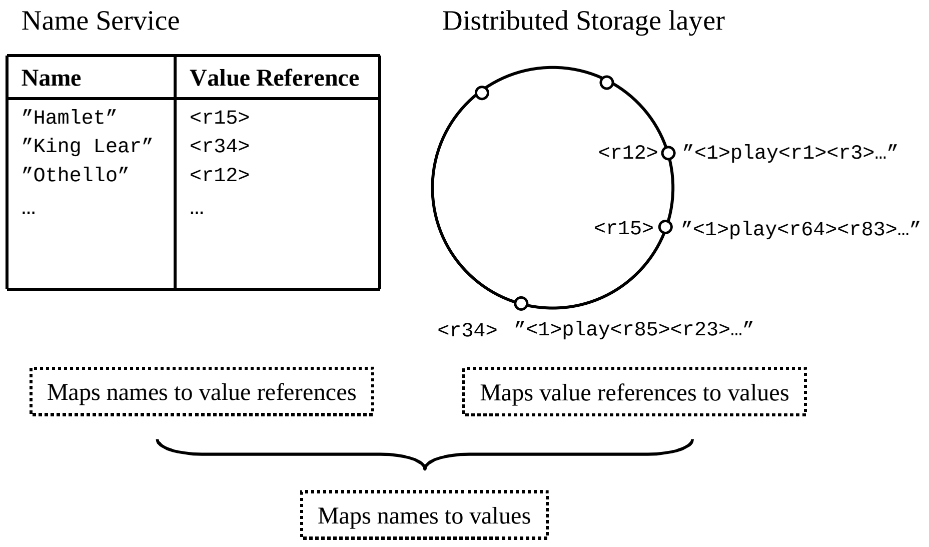

Name Service
Distributed Storage layer
Name
Value Reference
"Hamlet" <r15>
"King Lear" <r34>
"Othello" <r12>
… …
<r12> "<1>play<r1><r3>…"
<r15> "<1>play<r64><r83>…"
<r34> "<1>play<r85><r23>…"
Maps names to value references
Maps value references to values
Maps names to values

Figure 10.1: The Name Service and the Distributed Storage layer.

The underlying Chord identifier space (which also comprises all possible value references) is flat – it merely consists of numbers between 0 and $2^{160} - 1$. More structured naming schemes, such as hierarchical naming, can be imposed upon the system. The simple name service described, is probably insufficient for building complex distributed systems, but it nevertheless outlines the kind of interaction possible between a value-oriented XML store and a name service. The Directory also makes the entire system vulnerable as it represents a single point-of-failure: If the Directory for some reason stops responding, all applications using the name service will no longer work.

10.3 Concurrency problems

Updateable references accessible by multiple clients inevitably lead to concurrency problems. If the system stores critical data it is important that transactions by several clients are scheduled so that their effect is serially equivalent.

When a transaction manager is based on a non-value-oriented style of programming, the algorithms concerning concurrency and recovery can become very complex, especially if the transactions are distributed. If updates are allowed the involved variables have to either be locked while the update takes place, or the contents of the variables have to be copied and remembered so they can be restored (rolled back) if something goes wrong along the way.

The value-oriented approach makes it easier to implement a transaction manager. The current value referred to by a variable is not affected by the new value being built. As the former value always persists, implementing roll-backs is easy.

One of the most common problems is the lost update problem, where data reaches an inconsistent state due to a process ignoring the modification made by another process [13]. One way to achieve serial equivalence, is to use exclusive locks. This approach is known as pessimistic locking. It is not very efficient though, as access to a locked object requires the request to be suspended until the object is unlocked. Exclusive locks can furthermore result in deadlocks.

Another way to achieve serial equivalence can take advantage of the value oriented model. This approach is called optimistic locking, and is based on the assumption that the likelihood of two clients' transactions accessing the same value is low. Transactions are allowed to proceed as if there were no possibilities of conflict with other transactions until the client completes its task. If a conflict arises, then one or more transactions are aborted and will need to be restarted by the client. The optimistic concurrency control is not without flaws, though. The primary drawback of this method is the risk of starvation, where a process has its transaction aborted repeatedly.

Optimistic locking in the value oriented model can be implemented as follows: The state of the system is maintained by an updateable reference $stateRef$ to a value v. To update the value v means computing a new value v' and to commit this update consists of setting the reference $stateRef$ to the new value v', which can be done in a single atomic assignment operation. To abort the update, consists of doing nothing at all, because $stateRef$ still points to the original value. Pseudocode for the `update()` operation employing optimistic locking is shown in figure 10.2. Note that `map` is a standard hash map, mapping names to value references.

```
void update(name, newRef, expectedRef)
  currentRef = map.get(name)
  if (currentRef == expectedRef)
     map.remove(name)
     map.put(name, newRef)
  else
     throw ConcurrentUpdateException
```

Figure 10.2: Optimistic concurrency control in the XML Store.

10.4 A distributed name service

One of the primary problems with the proposed name service, the Directory, is that it makes the system vulnerable by introducing a single point-of-failure. In this section we will sketch how the Chord protocol can be used for implementing a decentralised name service. This is practical in relation to the XML Store as the Chord protocol is already implemented. The name service cannot be totally integrated with the existing XML Store though, as the name service requires updates to data.

The idea is to assign responsibility for a name to a peer, by taking the content hash of the name. By hashing a name we obtain a Chord identifier (a 160-bit key) and thereby we can assign a node the responsibility for storing information about a given name, using the Chord protocol's usual scheme for mapping keys to nodes. A binding – the association between a name and a document – consists of storing the Chord identifier obtained from hashing the name, paired with the value reference of the bound document.

If a given name is to be resolved, the name is hashed to a key and the node responsible for storing the current value of the binding is looked up using the Chord protocol, which can be accomplished in expected $O(\log N)$ hops. Consider a client looking to find the document containing the latest weather forecast. The name `xmls://dmi.dk/forecast.xml` is hashed to a Chord identifier, and the node responsible for storing the binding is looked up. The node returns the value reference of the document containing the current forecast. The client can now use the value reference obtained, to look up the actual document in the XML Store.

Problems related to updates

The fact that name bindings can be updated introduces complexities and pitfalls that must be taken into account. When storing the usual $\langle$value reference,value$\rangle$ pairs in the Distributed Storage layer, it is assumed that a key will only ever be associated with a single value. The key is a function of the value and different values will result in different keys. The deterministic relationship between key and value is what makes replication and caching simple to implement. However, in the case of a distributed name service, data is updateable. This makes the Chord caching strategy useless, unless the application can accept the risk of receiving an obsolete value reference.

The absence of caching can be tolerated, but it is unacceptable that a binding is lost if a peer stops responding. Therefore, replication must be implemented in a distributed name service, but this requires the replication strategy to be augmented with a coherence protocol.

Chapter 11

Network communication

In this chapter we discuss the implementation of the network communication in the XML Store network. We start by giving a brief description of *remote method invocation*[1] (RMI) and then analyse three different implementation strategies of the request-reply protocol, which RMI is based on.

11.1 Description of RMI

The XML Store network is based on a peer-to-peer system which means that each XML Store peer acts as both client and server. Because we use object-oriented programming to implement the XML Store, we choose RMI for communication between peers. RMI is an extension of local method invocation that allows an object living in one process to invoke the methods of an object living in another process possibly running in a different computer. The advantage of using RMI is that it provides network transparency in the sense that the syntax of a *remote* method invocation is almost the same as that of *local* invocation. The only syntactical difference is that a remote method invocation can throw an exception if the network communication goes wrong.

In RMI, the client (the requesting peer) references a *proxy* object, which behaves like a local object to the client. But instead of executing an invocation, it forwards it in a message to a remote object on the server peer holding the actual object. The request message is received by a so called *skeleton*, which unmarshals the arguments in the request message and invokes the corresponding method in the remote object. The skeleton waits for the invocation to complete and then marshals the result in a reply message to the sending proxy's method [5]. The role of the proxy and the skeleton is illustrated in figure 11.1, where Object A performs a remote method invocation on remote object B .

[1]Note that we use the term *RMI* to refer to remote method invocation in general, and not to a particular implementation, such as Java RMI.

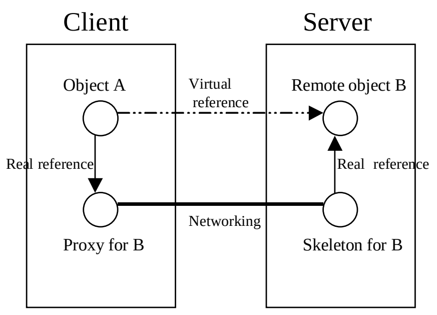


Figure 11.1: Illustration of a remote method invocation.

11.2 Synchronous message passing

RMI is based on the *request-reply* protocol [5], which is depicted in figure 11.2. Different kinds of implementations of this protocol exist, and we need to find the implementation that is best suited for the XML Store system. As illustrated in figure 11.2 the request-reply protocol uses synchronous communication: The client blocks until the reply arrives from the server.

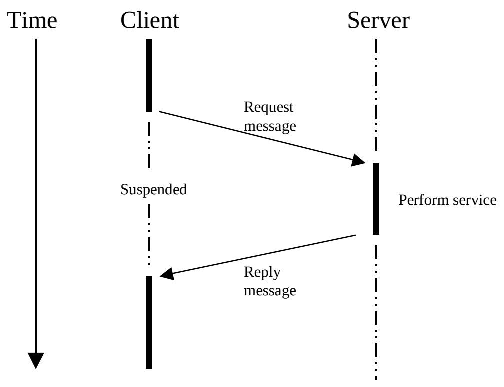


Figure 11.2: The request-reply protocol.

The main design question of the request-reply protocol is whether the server should use blocking or non-blocking I/O to read the incoming requests. In the non-blocking version, the server multiplexes asynchronous requests into a single thread. In other words, the server selects an eligible request from the set of incoming requests. This design allows the server to handle thousands of open connections while delivering scalability and high performance. However, some of the remote method invocations of the XML Store are recursive, meaning that a cycle of related invocations are possible. Having just one thread handling all incoming requests might therefore lead to a deadlock.

Instead of using just one thread to handle all incoming requests we need to allocate a separate thread for the execution of each remote invocation. Because of the overhead caused by thread management, this solution is not as scalable as the non-blocking solution.

To avoid that an unbounded number of threads are tied up because of missing reply messages or server crash, waiting threads have to time out. After a timeout the client peer can take appropriate action by for instance retrying the request.

11.3 RMI invocation semantics

As explained earlier, RMI is an natural extension of local method invocation. Local invocations are executed exactly once, but because of network failures this might not always be the case for RMI. There exist different kinds of semantics for the reliability of RMI as seen by the invoker.

With *maybe* invocation semantics, no fault tolerance is applied and the invoker cannot tell whether a remote method has been executed once or not at all [5]. This type of semantics can suffer from omission failures if the invocation or result message is lost and from crash failures when the server containing the remote object fails.

With *at-least-once* invocation semantics, retransmission of request messages is applied and the invoker receives either a result, in which case the invoker knows that the method was executed at least once, or an exception informing it that no result was received [5]. At-least-once invocation semantics can suffer from crash failures and from arbitrary failures, when the remote object executes the method more than once, because the invocation method is retransmitted.

With *at-most-once* invocation semantics, the invoker receives either a result, in which case the invoker knows that the method was executed exactly once, or an exception informing it that no result was received. In this case the method have been executed either once or not at all [5]. At-most-once invocation semantics can be achieved by using retransmission of request and results and filtering out duplicate requests at the server.

The XML Store use TCP/IP as the underlying network protocol and throws an exception if a network failure occurs. The RMI of the XML Store therefore has *at-least-once* invocation semantics.

11.4 Asynchronous message passing

Asynchronous request-reply communication is an alternative that is useful in situations where the client can afford to retrieve the reply later. Instead of waiting for the reply to arrive, the client proceeds independently of the server. This solution prevents deadlocks, since the client peer does not block and in addition allows the server peer to be non-blocking. Observe that the purpose of the waiting thread in the synchronous solution is to remember what to do with the result of the request.

The thread is simply a continuation for a particular request. Thus, instead of using threads, the asynchronous solution assigns a peer-unique identifier to each request made from the peer and keeps a table of outstanding requests: those that have been sent but for which a reply has not yet been received.

When a reply (with the associated request id) is received, the id and the corresponding action are located in the table and removed from it, and the action is applied to the reply. All replies could be received by a single thread, provided no actions involve blocking operations. Since the handling of a reply may involve sending a reply to another peer, replies must be non-blocking.

To implement timeouts on requests, an expiration time is associated with every request. A separate thread can periodically run through the table of outstanding requests and remove those that have expired. To do this in an proper way, a request may have a non-blocking method that can be called upon timeout.

The asynchronous solution can be implemented using UDP as the underlying network protocol. UDP saves network resources and avoids the expense of setting up connections.

11.5 Our choice of implementation

In the light of the above analysis, it should be clear that the third solution, the asynchronous message passing, is the solution that is best suited for the XML Store. However, the implementation of this solution is complex and error prone, and since the implementation of the XML Store is only a proof-of-concept prototype we have chosen to implement the synchronous thread solution. To have complete control of the network communication we do not use Java's RMI, but implement a solution that is custom-made to the XML Store system. The actual implementation and the classes involved are described in chapter 14.

Chapter 12

Security issues

Since any peer-to-peer system is a potential target for malicious attacks, security measures have to be taken into consideration. In the following we discuss the three classes of security threats mentioned by Coulouris et al. [5], namely *tampering*, *leakage* and *vandalism*, in relation to the XML Store system.

12.1 Tampering

Tampering concerns the alteration of data by unauthorised users [5]. In distributed systems, tampering constitutes a serious risk, due to the importance and intrinsic value that the data might have to the owners/users of the data. It is therefore important that vicious individuals cannot destroy data by alteration or corruption – or at least that tampering can be detected.

Since data that resides in the XML Store is identified according to its content hash, it is easy to detect whether data has been tampered. When one fetches a piece of data from the XML Store, one only needs to content hash the received data, and compare it to the content hash of the original data, to make sure that the data has not been altered. It is computational infeasible to tamper data, in such a way that the content hash remains the same as before the data was tampered [63].

Furthermore, distributing multiple replicas of data protects against tampering, besides making the system more robust. To destroy a given piece of data, a vicious user must obtain every replica of the data, which means that she needs to know every server where the data or replica of data is stored. Since a Chord peer cannot choose its own identifier (recall from section 8.1.1 that the identifier is a content hash of the peer's IP address), it cannot choose what data will be stored on it. Therefore, it is difficult to gain control over a peer that holds a particular piece of data, and thereby also difficult to tamper a specific piece of data.

12.2 Leakage

Leakage concerns the acquisition of data by unauthorised users [5]. When data is distributed among several peers, we need a way to secure the data from being read by unauthorised users, to provide a certain degree of confidentiality.

A solution to this confidentiality problem would be to encrypt the data before saving it in the XML Store. However, encryption of data could interfere with detection and coalescing of identical data, and thereby cause a loss of sharing. This is due to the fact that different users most likely use different keys to encrypt a given piece of data, which usually produces different cipher text.

To overcome the problems regarding detection and coalescing of identical files when encrypting files using different keys, we have devised a cryptographic technique called *Content Hash Encryption*, which is inspired by the *convergent encryption* technique, proposed by Bolosky et al. [67]. With this approach identical files will have identical cipher text.

12.2.1 Content Hash Encryption

The main idea of *Content Hash Encryption* is to use the content hash of a value as a kind of private key. The technique involves two content hashing steps – the first involving the data to be encrypted, the next involving the encrypted data. To encrypt a piece of data, d, by using *Content Hash Encryption*, five steps has to be performed. The encryption and decryption *functions* takes two parameters as input, namely the text which is going to be transformed (in our case d or e) and a *key* (in our case Key1 or Key2).

1. Content hash (d) = Key1
2. Encrypt (d , Key1) = encrypted data = e
3. Content hash (e) = Key2
4. Save (Key1 , Key2) pair locally
5. Save (Key2 , e) pair in the XML Store

To decrypt an encrypted piece of data, e, the user must know the content hash of the original data, i.e. Key1. Decryption of data, then involves the following step:

1. Load e using Key2
2. Decrypt(e , Key1) = d

Since Key1 has to be known to be able to decrypt data, data can be kept confidential in the XML Store system and at the same time, identical data can be detected and coalesced. The reasons for this are that first of all, Key1 is a one-way hash of the data, and therefore it is computational infeasible to guess the value of

Key1 – data is thereby secured from being read by unauthorised users. Secondly, identical data will get identical cipher text (e) when encrypted with Key1, since identical data will result in the same Key1. Therefore, Key2 will also be identical for identical data, since Key2 is a content hash of the cipher text.

To summarise, the *Content Hash Encryption* technique allows us to restrict data access via encryption while maintaining sharing of identical data. However, *Content Hash Encryption* is not yet an integrated part of the XML Store system.

12.3 Vandalism

Vandalism concerns the interference with the proper operation of a system [5]. An example of vandalism is denial of service attacks. In distributed systems, it is important to be able to prevent such *virus-style* attacks, for instance by disrupting the attacked service.

The XML Store system is, in its present state, quite sensitive to vandalism. Vicious users could insert a large amount of data into the system, thereby using up all the disk space on the XML Store peers. If all disks were full, there would be no space available for genuine data.

In the PAST system [3, 4] and CFS system [2], quotas are used to restrict the amount of data that a publisher is allowed to publish. This, however, requires some sort of reliable identification of publishers. In the PAST system, smartcards are used to provide this identification of publishers, whereas the CFS system base quotas on the IP address of the publisher. Quotas are not a part of the XML Store system yet. Other security measures still need to be considered to prevent vandalism of the XML Store system.

Chapter 13

Summary of the analysis

This chapter summarises the theoretical and conceptual properties of the designed XML Store and compare them to the desiderata from section 1.3.

Properties of the XML Store

The XML Store provides a simple and efficient API for building, manipulating, storing and retrieving XML documents. The API models XML documents as an object structure, allowing the programmer to work with an abstract tree representation of the document instead of a flat, serial text file. It allows the programmer to traverse and address parts of an XML document and provides various methods for modifying the document. The API is value-oriented, meaning that sharing of identical subdocuments is encouraged and all modifications to a document result in a new document containing the changes with the old document still intact. All the core operations of a child list of size n can be accomplished in time $O(\log n)$, except the `indexOf()` operation which has a worst case run time of $O(n)$. The API uses an abbreviated syntax of XPath, which is used in utility methods for modifying entire XML documents, thereby relieving the application programmer from modifying large XML documents manually.

The distributed storage layer of the XML Store is based on the Chord protocol, which allows $\langle$value reference,value$\rangle$ pairs to be stored and retrieved in a distributed peer-to-peer system. The Chord protocol is highly scalable since the lookup operation operates in time logarithmic in the number of servers and only requires logarithmic routing information at each peer. Furthermore, by using the Chord protocol the XML Store becomes decentralised, self-organising and provides some degree of load balancing.

We apply a cryptographic hash function as the value reference of a value, and we thereby obtain a value reference that is both deterministic, injective, hard to forge and with a very high probability unique.

When saving a document we split it up according to its inherent tree structure and store each node separately. This way we achieve a tree structured storage format on disk. Because of this storage strategy and because we work with immutable values, sharing of subdocuments is possible. Sharing makes updating of already stored documents very efficient since only the changed parts need to be stored, not the entire document. Furthermore, due to the storage strategy we can employ lazy loading when loading a document, and we thereby avoid the necessity to load and parse the entire document when accessing parts of it.

Value-oriented programming in a distributed environment

The value-oriented programming model used by the XML Store presents simple solutions to central problems of distributed systems.

Easy replication and caching Working with immutable data requires no cache coherence protocol and the immutable data can be freely replicated and coalesced without the need of complex replication protocols.

Atomic updates Since value-oriented programming keeps both the updated and the original version when data is modified the atomic update property of transactions is easily accomplished and recovery algorithms are not necessary.

The XML Store properties compared to the desiderata

The XML Store has the following desired properties:

- *No single point of failure*
- *Scale gracefully*
- *Load balance*
- *Self-organising*
- *Simple*
- *Efficient XML processing*
- *Applicable API*

By building the XML Store on top of a peer-to-peer system based on the Chord protocol we automatically achieve the desired properties: *No single point of failure*, *Scale gracefully*, *Load Balance* and *Self-organising*. Presently, the XML Store are not *Fault tolerant* and *Available* since replication and successor lists are not yet implemented. However, the XML Store could easily be augmented to contain these features, as done in the CFS system [2].

During the design of the XML Store we have given high priority to the central properties of peer-to-peer systems, namely decentralisation and scalability. In theory, the performance of the storing and retrieval operations of the XML Store is therefore poor compared to the performance of reading from and writing to local disk. The proposed storage strategy is efficient for loading large documents and updating already stored documents. However, due to the network overhead associated with a save operation, it is highly inefficient to store each node separately. Since both the XML Store and CFS system [2] are based on the Chord protocol, the XML Store should be able to achieve the same performance as CFS, which has download speeds competitive to FTP. However, with the present storage strategy such a performance is presumably not possible and another storage strategy should be employed as suggested in section 9.3.3. To achieve the desired performance, another implementation of the network communication is also necessary, e.g. asynchronous request-reply communication based on the UDP protocol.

A well-designed heuristic for accessing peers according to network proximity might speed up network access. In contrast to Chord, several routing and location schemes, such as Plaxton [45], Tapestry [47] and Pastry [22], attempt to approximate real network distance. It is possible to augment the XML Store system with a network locality heuristic as done in the CFS system.

As a consequence of working value-oriented the XML Store is *Simple* and has *Efficient XML processing*. The latter property is also acquired because we use a value-oriented red-black tree to represent the child list of an XML element. Furthermore, by providing a value-oriented API that is both simple and as versatile as DOM the XML Store obtain the desired property of an *applicable API*.

Security issues are not implemented in the XML Store. However, in chapter 12 we presented an encryption strategy, Content Hash Encryption, that makes it possible to encrypt values and still maintain sharing, thereby preventing unauthorised user from reading the contents of values (*Leakage*). Furthermore, it is difficult to tamper data that resides in the XML Store in a way that makes it hard to detect whether the data has been tampered (*Tampering*). However, we still need to consider security measures for preventing *Vandalism*.

The XML Store does not have a search facility nor does the it support queries.

Part III

Implementation & evaluation

Chapter 14

Implementation

In this chapter we describe the overall structure of the prototype implementation of the XML Store. The purpose of this section is not to describe the code in detail, but to grant the reader an overview of the program. The diagrams will not include all classes and the depicted classes will not feature all fields and methods – only parts central to the understanding of the system are shown. The diagrams conform to the UML standard, but as there exists many different "flavours" of UML, especially when depicting methods, we briefly describe the subtleties in table 14.1.

Initially we describe how the layers introduced in chapter 6 are implemented. We then describe the implementation of the network communication and finish by giving an example of how an application programmer can build a system using the XML Store and associated XML API. The source code can be found in the appendix as well as on the web at
`http://www.it-c.dk/people/fenne/xmlstore/src.zip`.

14.1 The XML Store Layers

As previously mentioned, the XML Store can be regarded as consisting of various layers, each responsible for a specific set of tasks. Note that the layers are a conceptual model – the current XML Store implementation is not entirely layered. The Distributed Storage layer and XML Storage layer are for instance tied together using inheritance. We still present the program in the context of these layers, to help

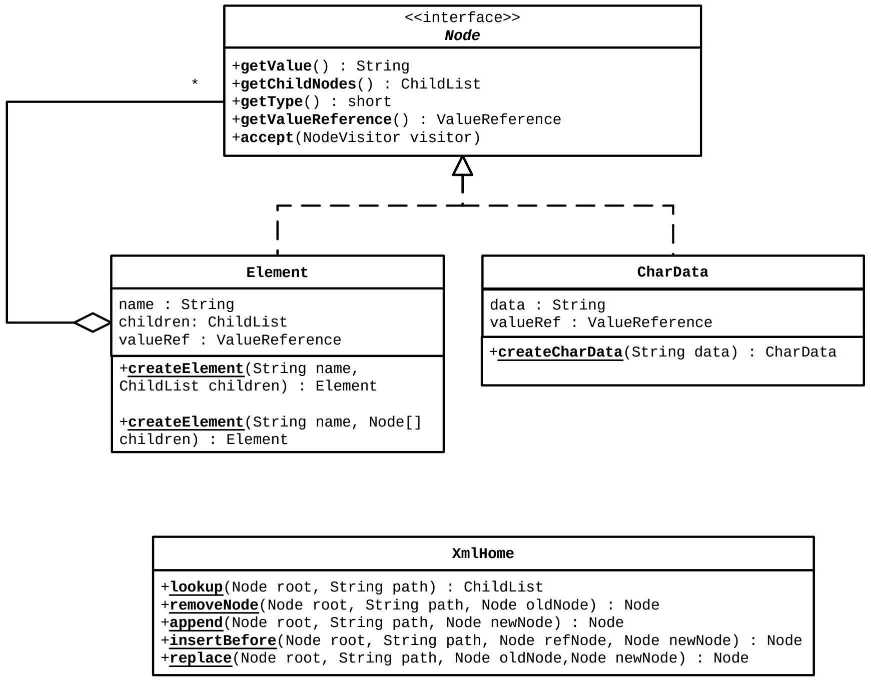
Syntax	**Meaning**
+	`public`
-	`private`
underline	`static`
italics	`abstract` or `interface`

Table 14.1: Our adaption of syntax for methods in UML-diagrams.

clarify the responsibilities of the classes and modules of the system.

14.1.1 The XML Layer

The classes comprising the XML layer are placed in package `edu.it.xmlstore.xml`. Most classes represent parts of an XML document – these include interface `Node` and its derived classes `CharData` and `Element`, as shown in figure 14.1. They offer methods for manipulating XML documents in a value-oriented way.



<<interface>>
Node
+getValue() : String
+getChildNodes() : ChildList
+getType() : short
+getValueReference() : ValueReference
+accept(NodeVisitor visitor)
*
Element
name : String
children: ChildList
valueRef : ValueReference
+createElement(String name, ChildList children) : Element
+createElement(String name, Node[] children) : Element
CharData
data : String
valueRef : ValueReference
+createCharData(String data) : CharData
XmlHome
+lookup(Node root, String path) : ChildList
+removeNode(Node root, String path, Node oldNode) : Node
+append(Node root, String path, Node newNode) : Node
+insertBefore(Node root, String path, Node refNode, Node newNode) : Node
+replace(Node root, String path, Node oldNode,Node newNode) : Node

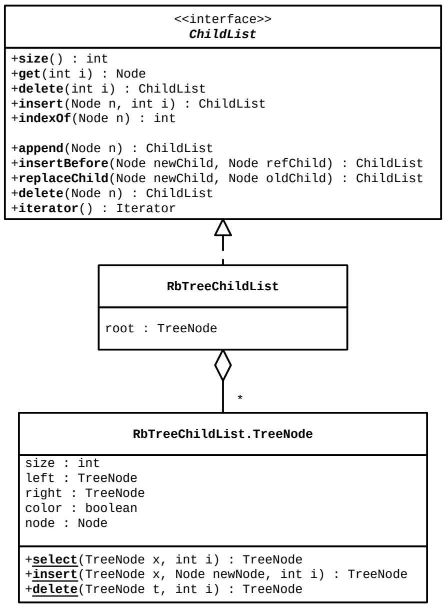
Figure 14.1: `Node`, `Element` and `CharData` represent (parts of) XML documents. This is an example of the Composite design pattern.

`CharData` and `Element` cannot be instantiated using the `new` keyword. They are instead created using `Element.createElement()` and `CharData.createCharData()`. This allows us to employ hashed consing to never instantiate two identical objects, thereby ensuring maximal sharing.

The children of an element are represented by interface `ChildList` and its subclass `RbTreeChildList`. `RbTreeChildList` contains private inner class `RbTreeChildList.TreeNode`, that represents the child list as an approximately balanced red-black tree. This is shown in figure 14.2.

The layer also provides utility methods for parsing XML files in flat text format to object representation (`Element.createElementFromFile()`), and externalise object representation of XML document to text format (`Node.asString()`).

The XML documents can of course be manipulated manually by traversing the node structure, adding, deleting or replacing nodes and building a new document accordingly. To make common tasks easier we offer a number of utility methods for modifying whole documents. These are collected in class `XmlHome`.



<<interface>>
ChildList
+size() : int
+get(int i) : Node
+delete(int i) : ChildList
+insert(Node n, int i) : ChildList
+indexOf(Node n) : int
+append(Node n) : ChildList
+insertBefore(Node newChild, Node refChild) : ChildList
+replaceChild(Node newChild, Node oldChild) : ChildList
+delete(Node n) : ChildList
+iterator() : Iterator
RbTreeChildList
root : TreeNode
*
RbTreeChildList.TreeNode
size : int
left : TreeNode
right : TreeNode
color : boolean
node : Node
+select(TreeNode x, int i) : TreeNode
+insert(TreeNode x, Node newNode, int i) : TreeNode
+delete(TreeNode t, int i) : TreeNode

Figure 14.2: `ChildList`, `RbTreeChildList` and `RbTreeChildList.TreeNode` represent the children of an element.

14.1.2 The XML Storage layer

The interface `XmlStoreServer` and its implementation `XmlStoreServerImpl`, in package `edu.it.xmlstore`, represents a single peer in the XML Store system. The `XmlStoreServer` allows the client to save and load XML documents from the XML layer and is responsible for all interaction with the Distributed Storage layer. The `XmlStoreServer` implements the storage strategy, deciding in which way to transform a given document to one or more byte-sequences to be stored. Figure 14.3 shows the interaction between `XmlStoreServer`, `XmlStoreServerImpl` and `XmlStoreServerHome`.

The class `XmlStoreHome` is a helper class containing methods to get the system started. It provides methods for making initial contact with another server and locating the name service, both by using IP multicast. See section 14.4 for a more detailed description of how to use `XmlStoreHome`.

A value reference is represented by the simple interface `ValueReference`, and is used throughout all layers of the system, except for the Distributed Storage layer. The actual implementation of `ValueReference` is `ChordIdImpl`, see section 14.1.3.

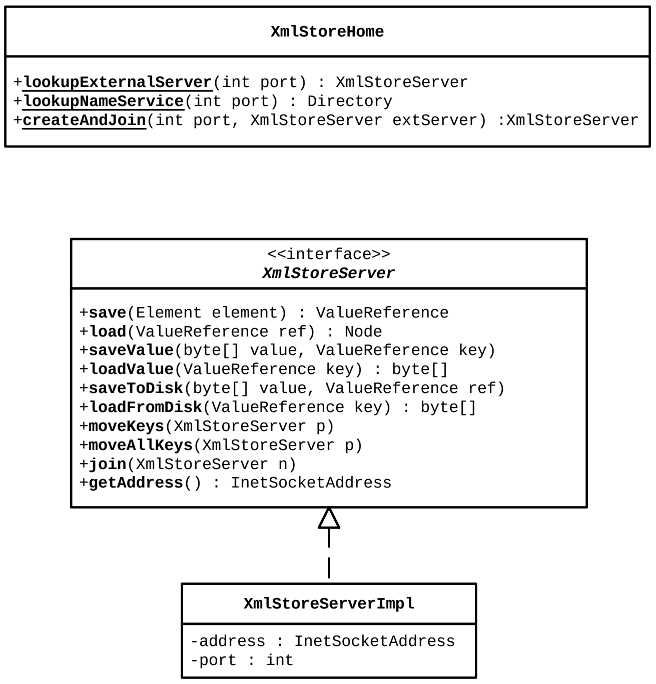
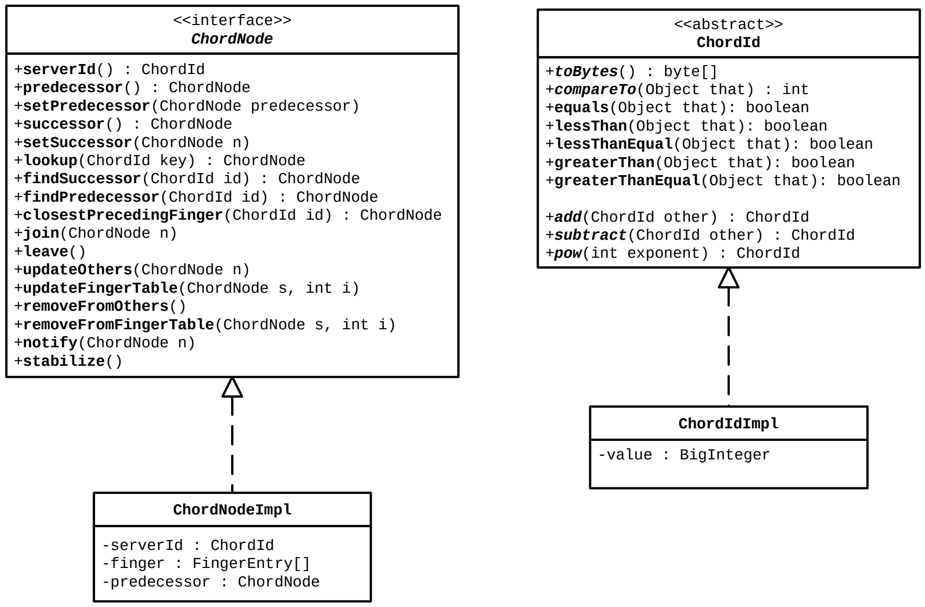

XmlStoreHome
+lookupExternalServer(int port) : XmlStoreServer
+lookupNameService(int port) : Directory
+createAndJoin(int port, XmlStoreServer extServer) :XmlStoreServer
<<interface>>
XmlStoreServer
+save(Element element) : ValueReference
+load(ValueReference ref) : Node
+saveValue(byte[] value, ValueReference key)
+loadValue(ValueReference key) : byte[]
+saveToDisk(byte[] value, ValueReference ref)
+loadFromDisk(ValueReference key) : byte[]
+moveKeys(XmlStoreServer p)
+moveAllKeys(XmlStoreServer p)
+join(XmlStoreServer n)
+getAddress() : InetSocketAddress
XmlStoreServerImpl
-address : InetSocketAddress
-port : int

Figure 14.3: `XmlStoreServer`, `XmlStoreServerImpl` and `XmlStoreServerHome` comprise the XML Storage layer.

14.1.3 The Distributed Storage layer

The Chord protocol is implemented in class `ChordNodeImpl` derived from interface `ChordNode`, both in package `edu.it.xmlstore.chord`. This class contains the basic Chord functionality such as joining and leaving a Chord ring, and looking up a node responsible for an identifier.

The Chord identifiers are represented by classes `ChordId` and `ChordIdImpl`. These classes encapsulate the handling of 160-bit integers and makes sure that all arithmetic is modulo 2^{160}. Value references and Chord identifiers are almost alike, but we nevertheless have different interfaces for them, though both interfaces are implemented by the same class. This is due to some subtle differences: A value reference only needs to be checked for equality, whereas a Chord identifier needs to be able to handle arithmetic with other Chord identifiers as well as more detailed comparison operators. The `ChordId` interface is used instead of value references in the Distributed Storage layer.



<<interface>>
ChordNode
+serverId() : ChordId
+predecessor() : ChordNode
+setPredecessor(ChordNode predecessor)
+successor() : ChordNode
+setSuccessor(ChordNode n)
+lookup(ChordId key) : ChordNode
+findSuccessor(ChordId id) : ChordNode
+findPredecessor(ChordId id) : ChordNode
+closestPrecedingFinger(ChordId id) : ChordNode
+join(ChordNode n)
+leave()
+updateOthers(ChordNode n)
+updateFingerTable(ChordNode s, int i)
+removeFromOthers()
+removeFromFingerTable(ChordNode s, int i)
+notify(ChordNode n)
+stabilize()
ChordNodeImpl
-serverId : ChordId
-finger : FingerEntry[]
-predecessor : ChordNode
<<abstract>>
ChordId
+toBytes() : byte[]
+compareTo(Object that) : int
+equals(Object that): boolean
+lessThan(Object that): boolean
+lessThanEqual(Object that): boolean
+greaterThan(Object that): boolean
+greaterThanEqual(Object that): boolean
+add(ChordId other) : ChordId
+subtract(ChordId other) : ChordId
+pow(int exponent) : ChordId
ChordIdImpl
-value : BigInteger

Figure 14.4: `ChordNode` and `ChordNodeImpl` comprise the Distributed Storage layer, making use of `ChordId` and `ChordIdImpl` to represent Chord identifiers.

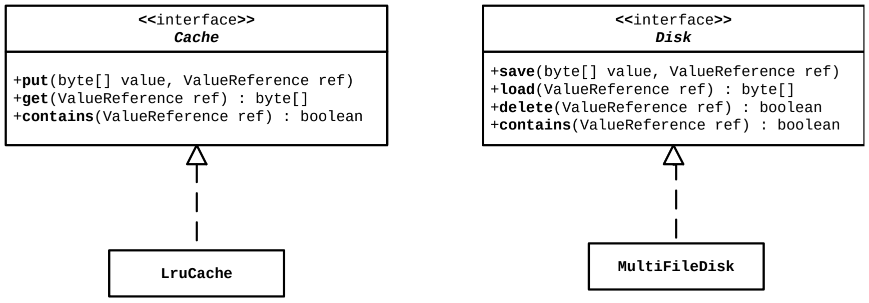
14.1.4 Storage layer

The Storage layer consists of the classes in package `edu.it.xmlstore.storage`, and is responsible for storing values locally (see class diagram in figure 14.5). The interface `Disk` and its derived class `MultiFileDisk` store values permanently on disk. The `Disk` interface simply maps value references to values – the client has no influence on how the files are stored or where. More efficient disk handling can easily be incorporated into the system by adding another implementation of the `Disk` interface.

The layer also has interface `Cache` and derived class `LruCache` for temporary caching of values. `LruCache` uses the least-recently-used strategy to determine which values should be deleted when the cache is filled up.

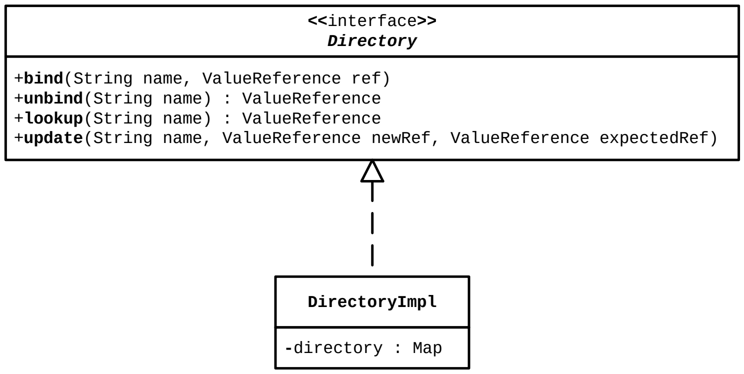
14.1.5 Name service

The Name Service cannot really be considered a layer, as it exists as an external service to the XML Store. The XML Store never uses the Name Service – it is the client that employs the service, to associate value references with human-readable names. The simple proposed name service has interface `Directory`, is implemented in subclass `DirectoryImpl` and summarised in figure 14.6. All classes related to the directory are located in package `edu.it.xmlstore.directory`.



<<interface>>
Cache
+put(byte[] value, ValueReference ref)
+get(ValueReference ref) : byte[]
+contains(ValueReference ref) : boolean
<<interface>>
Disk
+save(byte[] value, ValueReference ref)
+load(ValueReference ref) : byte[]
+delete(ValueReference ref) : boolean
+contains(ValueReference ref) : boolean
LruCache
MultiFileDisk

Figure 14.5: `Disk` and `MultiFileDisk` provides persistent storage, `Cache` and `LruCache` provide temporary caching.



<<interface>>
Directory
+bind(String name, ValueReference ref)
+unbind(String name) : ValueReference
+lookup(String name) : ValueReference
+update(String name, ValueReference newRef, ValueReference expectedRef)
DirectoryImpl
-directory : Map

Figure 14.6: `Directory` and `DirectoryImpl` constitutes a simple name service.

14.2 XmlStoreServer

This section will describe in more detail the classes that make up an XML Store server. The class `ChordNodeImpl` contains only the functionality required for handling joining, leaving and looking up a node given an id. The `XmlStoreServerImpl` extends `ChordNodeImpl` as shown in figure 14.7, gaining access to the routing algorithms and supplying such functionality as storage of values. Where needed, e.g. when joining and leaving, `XmlStoreServerImpl` overrides the original methods to add special behaviour. Other applications wishing to use the Chord routing algorithm can likewise extend class `ChordNodeImpl`.

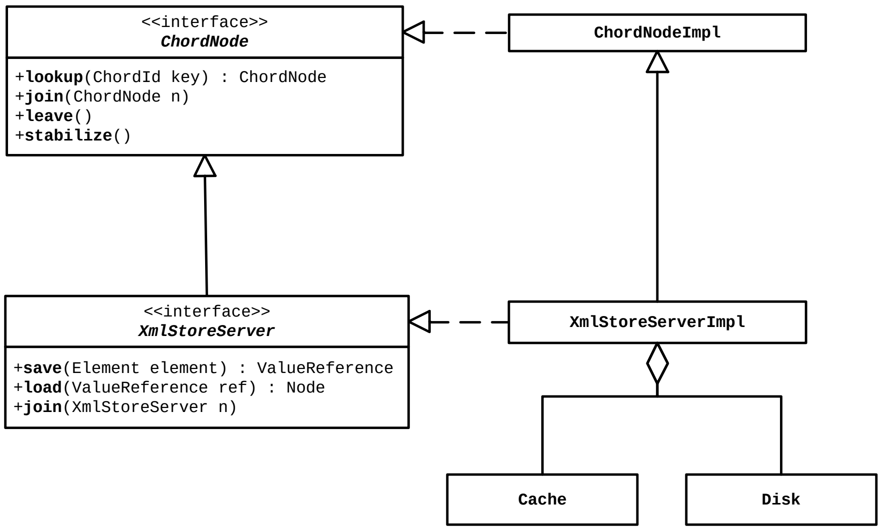


Figure 14.7: Class diagram illustrates the relationship between `XmlStoreServer` and `ChordNode`.

14.3 Network communication

Communication from peer to peer is essential to the system and very complex due to the recursive nature of the Chord protocol. The design of the communication between peers is one of the most complicated parts of the program. It has substantial influence on the overall performance of the XML Store system.

The design is inspired by RMI as described in section 11.1 and relies heavily on the use of proxies, abstracting the actual network communication from the application programmer. We have emphasised that the design uses interfaces heavily, and designed the interfaces so it is easy to extend the system with more efficient implementations.

We use the classes in package `edu.it.xmlstore.rpc`, and the relationship between these classes is shown in figure 14.8.

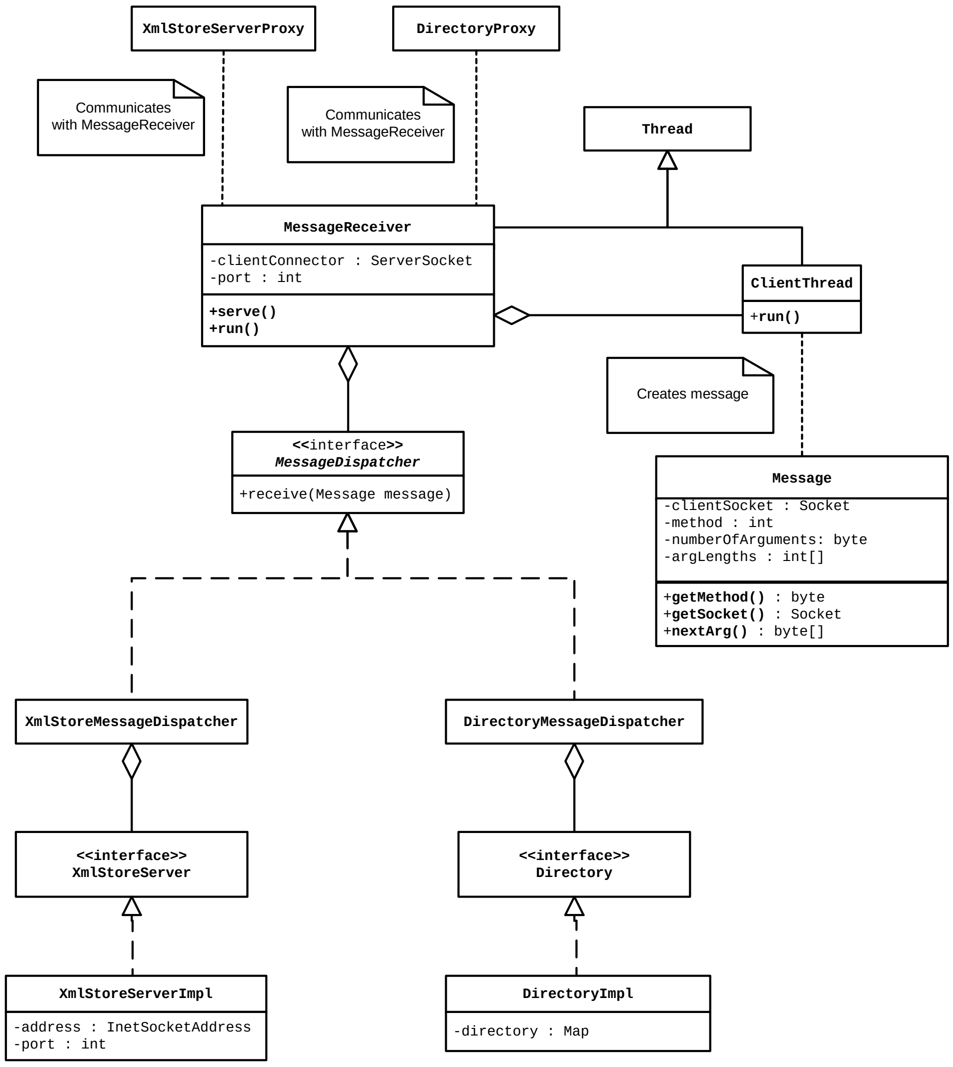


Figure 14.8: The classes used for network communication.

In figure 14.9 we present a sequence diagram of a single remote method invocation to give an overview of the interaction between the involved classes. The figure depicts the situation where `XmlStoreServerImpl` A wants to invoke the method `findSuccessor(id)` on `XmlStoreServerImpl` B. The remote method invocation is performed by A calling a proxy, `XmlStoreServerProxy` for B. The proxy then marshals the argument, `id`, and forwards the method invocation to the real B using a socket connection. The call is received by a `MessageReceiver`, which creates a new instance of the `ClientThread` class for each incoming request. When the `ClientThread` has received the entire request it creates a new `Message` object and passes it on to the `XmlStoreMessageDispatcher`. The `XmlStoreMessageDispatcher` performs the role of the skeleton, as described in section 11.1. It determines the type of method, unmarshals the argument and calls the `findSuccessor(id)` method on `XmlStoreServerImpl` B. The `XmlStoreMessageDispatcher` then marshals the result and sends it back to `XmlStoreProxy`, which unmarshals the result and returns it to `XmlStoreServerImpl` A.

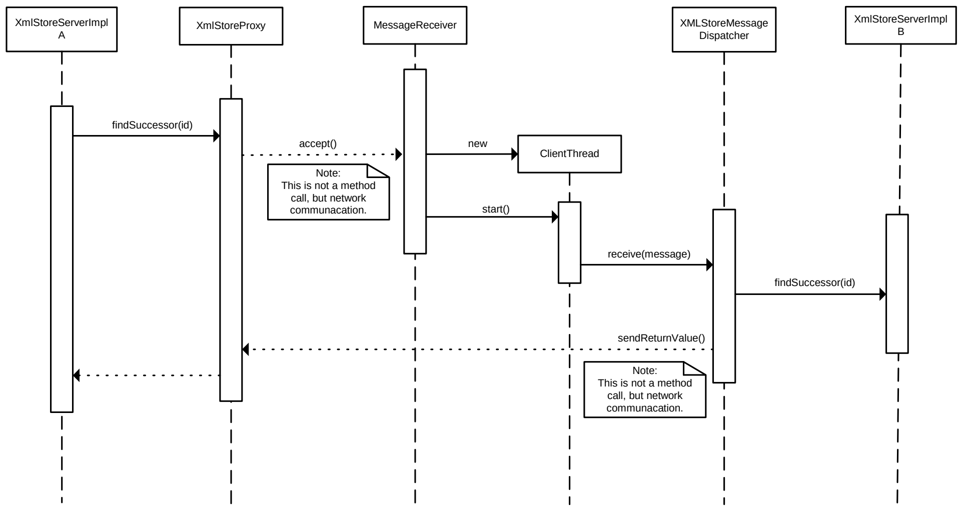


Figure 14.9: Sequence diagram of a remote method invocation.

Notice that the network communication between an XML Store and the name service (`Directory`) is handled in the same way as illustrated in the above sequence diagram. However, instead of using the `XmlStoreMessageDispatcher`, the name service uses `DirectoryMessageDispatcher` and `DirectoryProxy` to invoke the requested method on the actual `DicectoryImpl` object.

14.4 Building applications using the XML Store

This section will describe how to use the XML Store, from an application programmers view.

The client can create and manipulate XML documents by using the classes in the package `edu.it.xmlstore.xml`. Consider a client wishing to produce the following XML fragment, representing a famous line from a play by Shakespeare:

```
<SPEECH>
  <SPEAKER>
     KING RICHARD III
  </SPEAKER>
  <SPEECH>
     A horse! a horse! my kingdom for a horse!
  </SPEECH>
</SPEECH>
```

The above could be constructed in the following way:

```
   Node speaker = Element.createElement("SPEAKER",
             new Node[] {
             CharData.createCharData("KING RICHARD III")});
   Node line = Element.createElement("LINE",
             new Node[] {
             CharData.createCharData(
             "A horse! a horse! my kingdom for a horse!")});
   Node speech = Element.createElement("SPEECH",
             new Node[]{newSpeaker, newLine});
```

If the above XML fragment was stored in a regular text file called “doc.xml”, the document could be constructed using a method that parses the file to `Node` representation:

```
   Node speech = Element.createElementFromFile("doc.xml");
```

To store the document in the peer-to-peer network we need to start an XML Store server. For a client or an upper application layer the XML Store server is accessible through the interface `XmlStoreServer`, which defines methods such as `save()`, `load()`, `join()` and `leave()`. `XmlStoreHome` defines helper methods for initialising an XML Store server. Clients wishing to join the XML Store network simply call `XmlStoreHome.createAndJoin()` to have an `XmlStoreServer` instantiated and joined the peer-to-peer network. The document can then be saved. The following shows the above document being saved. Note that `port` can be any unused port in the system.

```
   int port = 6001;
   XmlStoreServer server = XmlStoreHome.createAndJoin(port);
   ValueReference ref = server.save(speech);
```

The client can associate the value reference with a human-readable name using the Directory. This makes the value reference available for later retrieval and makes the document available to other interested parties. The following code fragment binds the above document (actually the document’s value reference) to a name.

```
String name = "Quote of the day";
Directory directory = XmlStoreHome.getDirectory(port);
directory.bind(ref, name);
```

For a more extensive example of how to build applications using the XML Store, see the email application in package `edu.it.xmlstore.mail`.

Chapter 15

Test

This section briefly discusses possible test strategies and present the test strategy used when implementing the XML Store system and the distributed e-mail application. The focal point will be systematic software test.

15.1 Systematic test

A program is tested to substantiate that it is correct and satisfactory, meaning that there should not be any errors in the program. To substantiate whether a given program works properly, a systematic *internal* or *external* test of the program can be made.

An *internal test* (also known as *white-box* test or *structural* test) is used to discover "logical" errors in a program, i.e. errors in the structure of the program, such as sequences that are never executed etc. Thus, in an internal test, all parts of a program must be examined. This means that all methods must be called and within each method every loop (`for` and `while`) and branch (`if` and `switch`) must be executed. To accomplish this, the test suite should contain a sufficient amount of input data. To be able to check the actual output, the expected output must be specified for every input data set, so that it can be compared to the actual output. For large programs, an internal test may be very time consuming.

External test (also known as *black-box* test or *functional* test) is used to test the functionality of a program. That is, to make sure that the program works as it should. In an external test, the program is viewed as a black-box. The only thing of interest is input data and output data of the program – not the internal structure of the program. To make an external test of a program, a test data set consisting of "typical" and "extreme" input values and corresponding expected output values is constructed. The expected output values are then compared to the actual output values.

Internal tests and external tests are complementary. The two types of test are used for different purposes – one does not cover the other.

15.2 Automatic test

Automatic test is a way to ensure that a program is continually tested. This makes it easier to discover newly introduced errors, since the program is tested every time code has been added, deleted or modified. To be able to automate tests, each test should be isolated from the other tests. Thereby, one avoids that if one test fails, it does not cause other failures as well.

15.3 Test of the XML Store system

During the course of implementing of the XML Store system, we have used automatic unit tests to test each class of the system. Our unit tests can be considered an external test of the system. We have made test-suites for every class and (nearly) every method in the program. These test-suites contain typical as well as extreme input data and for each test case we have specified the expected output value. The test suites are automatically run every time the program is compiled, which makes it easier to discover newly introduced errors. The test suites can be found in the appendix.

Even though we have tested the system quite thoroughly using external tests, we cannot exclude the possibility that there might still exist errors in the system. To be absolutely certain that the system has no errors and works 100% correct, we would have to *prove* the correctness of the system for every possible input. However, this strategy is infeasible for anything, but very small programs.

We have not conducted a dedicated system test of the XML Store in its entirety. However, we have made quite a lot of successful experiments of the XML Store and have built an e-mail application on top of the XML Store. We are therefore convinced that the XML Store system works.

Chapter 16

Experimental results

In order to evaluate the performance of the XML Store design we present four sets of experiments. The first experiment concerns the performance of storage and retrieval of documents. The second experiment focuses on the scalability of the system. The third experiment seeks to evaluate the value-oriented storage strategy, whereas the fourth experiment explore the efficiency of the XML Store API implementation. Where relevant, we identify the causes for poor performance and suggests strategies for boosting performance. Notice that we do not examine the properties of the Chord protocol, since this has already been done by Stoica et al. [42].

16.1 Performance of storage and retrieval

To determine the performance of storage and retrieval of documents, we set up a network of 50 XML Store peers distributed over five physical machines[1]. A number of XML documents were saved in the XML Store network and then loaded. The configuration of the XML documents vary over two parameters: The number of nodes and the size of the character data nodes. As each node is saved separately the depth of the tree does not affect the performance. The times listed are the average of three separate experiments.

The XML Store employs a lazy loading strategy by using proxy nodes, that only load the actual node if it is required. In the experiment we make sure the document is completely loaded, by traversing the entire node structure and accessing the content of each node.

Because of the storage strategy of the XML Store, where each node is saved separately, we expect the performance of the load and save operation to be dominated by the number of nodes in the XML documents and only to a lesser degree, by the size of the nodes. Furthermore, we expect the load operation to be a small

[1]IBM compatible PCs. Equipped with AMD 1800 MHz CPU, 256 MB RAM, 40 GB IBM DeskStar HD. Connected by 10 Mb ethernet. Running Windows XP Pro with Sun JVM 1.4.0 using (default) 32 MB heap size.

constant factor faster than the save operation, since reading from disk is normally faster than writing to disk. Note that the XML Store's caching scheme does not influence these readings, as only situations where different peers load the same value are affected – not situations where a single peer loads different values.

16.1.1 Influence of number of nodes

The three graphs represented in figure 16.1 illustrate the influence of the number of nodes in an XML document, on the time it takes to save and load the XML document. The XML documents used for testing, consist of 1 to 10000 nodes, each containing character data of either 30, 300 or 3000 bytes.

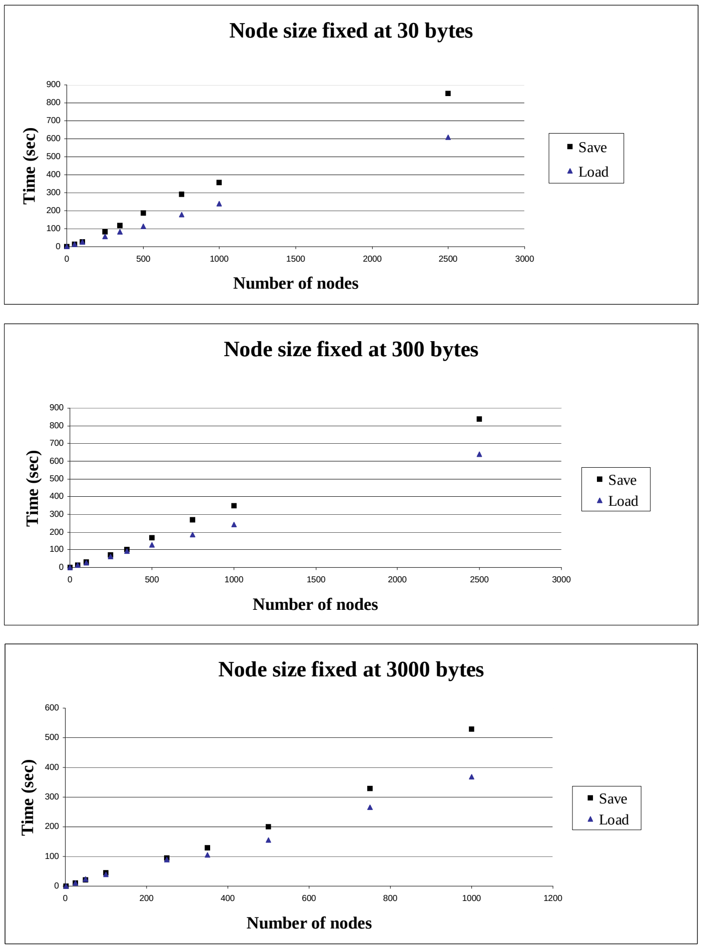


Figure 16.1: Influence of the number of nodes in an XML document on the time it takes to save and load the document. The three graphs represent nodes containing character data of either 30, 300 or 3000 bytes. Results are the mean of three independent test runs. The test runs did not exhibit large variations.

As expected, the graphs clearly show that the running time of both save and load exhibits linear behaviour as the number of nodes increases. Furthermore, the results show that, as expected, it is faster to load a document, than to save it. The graphs also show that the time it takes to save and load a document tend to increase as the size of the character data that a node contains, grows. In the following, the influence of the size of character data on save/load times is examined.

16.1.2 Influence of the size of character data

The three graphs represented in figure 16.2 show the influence of the size of character data that a node contains, on the time it takes to save and load an XML document. The test XML documents consist of a fixed number of nodes (either 10, 100 or 1000 nodes) and each node contains between 5 and 12500 bytes of character data.

The results of this experiment show that the size of character data has some influence on the time taken to both save and load an XML document. The time taken when saving and loading a document grows with the size of character data in each of the three cases, although very slowly and at times somewhat randomly. Especially in the experiments involving relatively few nodes (documents with 10 and 100 nodes), the time taken to save the smaller documents exhibits strange behavior, being somewhat slow. We have no explanation for this phenomenon, other than inaccurate measuring and unreliability caused by the relatively small number of nodes. The experiment involving a large number of nodes does not exhibit these deviations. Again, as expected, it is faster to load a document than to save it.

16.1.3 Number of nodes vs. size of character data

To estimate the influence of the number of nodes contra the size of character data on performance, we compare XML documents of approximately equal size, but with a varying number of nodes and hence also varying sizes of character data (see table 16.1).

As can be seen from table 16.1, there is a significant difference in the time it takes to save XML documents of approximately the same size, but which vary in the number of nodes. The more nodes, the longer it takes to save the XML document. Since the documents are of roughly equal sizes, we consider the number of nodes as the predominant factor regarding the time it takes to save an XML document. This is understandable since each node is stored separately requiring expensive network communication.

16.1.4 Network communication

The XML Store stores roughly between $1 - 4$ nodes per second which is inefficient and unsatisfactory. The cause of this is first and foremost the implementation of

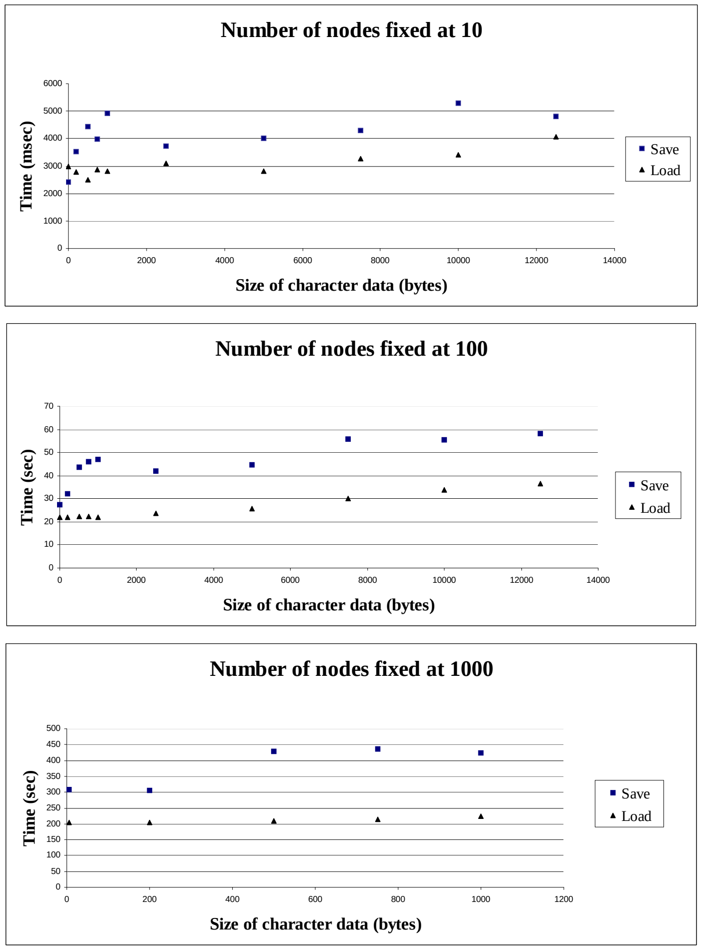


Figure 16.2: Influence of the size of character date (bytes) that nodes in an XML document contain, on the time it takes to save and load the document. The test XML documents consist of a fixed number of nodes – either 10, 100 or 1000, corresponding to the three graphs. Results are the mean of three independent test runs. The test runs did not exhibit large variations.

Size of document (kB)	Number of nodes	Size of character data (bytes)	Time (sec)
493	100	5000	45
531	1000	500	429
743	250	3000	96
736	10000	30	2892
1041	350	3000	128
1179	3500	300	1006
1486	500	3000	201
1685	5000	300	1507

Table 16.1: Comparison of pairs of XML documents of approximately the same size, with regard to number of nodes, size of character data and the time it takes to save the documents.

network communication. As described in chapter 11 we use TCP-based communication and rely on threads to handle simultaneous requests to a server. It is a simple solution that can handle the recursive nature of the Chord protocol, but it is very expensive to handle each incoming request in a thread of its own.

The speed at which messages can be exchanged between peers in the system constitutes an upper bound to the efficiency of the XML Store. If the network communication is not improved it will be very hard to make the XML Store perform better. The network communication could be improved e.g. by using a thread pool instead of creating new threads for each message. Another improvement is to use connectionless UDP communication instead of TCP, but that necessitates that network error handling is performed manually.

16.1.5 Asynchronous storage of nodes

Another possibility for substantial improvement is to exploit the system's inherent parallelism. As each node is an independent entity it can be saved independently of the other nodes in the document. There is no need to await the confirmation of a saving of a node, before saving the rest of the document. As the nodes comprising a document with high probability are stored at different peers, the time spent waiting for one peer in the system to complete a save operation could be put to better use by saving several nodes simultaneously.

16.2 Scalability

The purpose of this experiment is to evaluate the scalability of the XML Store system. In other words, to find out if the system becomes "hopelessly slow" when many XML Store peers have joined the peer-to-peer system.

The experiment is performed by saving a number of identically structured and sized, but nonetheless different, XML documents in a network with an increasing number of XML Stores. Each of the save operations is timed and the results can be seen in figure 16.3.

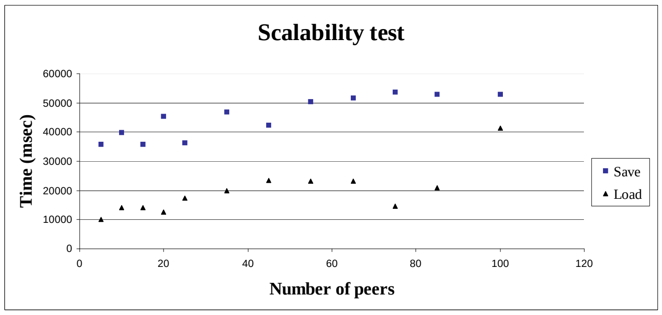


Figure 16.3: The influence of the number of peers in the system on the time taken to save a document. Results are the mean of three independent test runs. The test runs did not exhibit large variations.

The results show that there are some variations in the results. It is thus difficult to determine whether there is a correlation between the number of peers and the time it takes to store an XML document.

Even though the experiment seems to suggest that the XML Store system will remain scalable as more peers join the network, the number of peers is nowhere near the magnitude that the system is supposed to handle in a real life situation. Thus, we cannot from this experiment conclude that the XML Store system scales gracefully when the number of peers increases, but rather have to rely on experiments done by Dabek [2]. Dabek has carried out a number of experiments, regarding the CFS system based on the Chord protocol, and these experiments show that the CFS system is highly scalable.

16.3 Modification of an XML document

The purpose of this experiment is to evaluate the value-oriented storage strategy when working with XML documents. Since the value-oriented data handling of the XML Store allows maximal sharing of subdocuments, we expect that modifying existing XML documents is considerably faster when working value-oriented, than when updates are allowed.

The experiment is performed by first creating and saving a large XML document ("The Tragedy of Hamlet, Prince of Denmark", by Shakespeare) in the XML Store. The document is then updated by either replacing, appending or removing nodes and saved in the XML Store after each update.

Saving an XML document using conventional technologies, such as flat text files, will result in a linear runtime of the save operation in the size of the document. If instead an XML document is saved using a value-oriented approach, and thereby

Operation	**Time (msec)**
Save “Hamlet”	773300
Add character	917
Modify title	341
Add speech	822
Modify line	831
Delete (final) scene	306

Table 16.2: Results of saving an XML document repeatedly after modifying it in various ways. Results are represented in milliseconds and are the mean of two independent test runs. The test runs did not exhibit large variations.

taking full advantage of sharing, the time it takes to save a document will be linear in the size of the *changes* made to the document.

Table 16.2 shows the results of saving the XML document “The Tragedy of Hamlet, Prince of Denmark” repeatedly in the XML Store, after modifying it in various ways. As can be seen from the results, it takes quite some time to save the XML document the first time. Saving the document again, after modifying it in different ways, is much faster, thereby demonstrating the advantages of using a value-oriented approach when saving documents. Furthermore, the value-oriented approach ensures that all versions of the document still exists in the XML Store.

16.4 Representation of child lists

The purpose of this experiment is to determine the efficiency of the child list implementation, thereby obtaining some information on the size of XML documents which can be handled in the XML Store. We experiment with two different child list representations, namely arrays and red-black trees. The experiment was performed by gradually inserting a number of nodes (5 to 100000 nodes) at random places into a child list represented either as an array or as a red-black tree. The results of the test can be seen in table 16.3.

From the results presented in table 16.3 it is obvious that the red-black representation is faster than the array representation, when inserting a large number of nodes. As can be seen, it takes approximately 2 seconds to insert 100000 nodes in a child list represented as a red-black tree. Therefore, we find that our implementation of the red-black tree representation of children is suitable for processing very large XML documents.

Number of nodes inserted	**Time (msec) using array**	**Time (msec) using R/B tree**
5	0	0
10	0	0
50	1	3
100	0	2
500	11	1
1000	41	5
1500	94	8
2000	164	11
5000	1071	37
10000	4518	116
15000	11421	228
25000	61106	405
50000	364993	923
75000	967788	1504
100000	1813750	2073

Table 16.3: Results of inserting a number of nodes randomly into a child list, represented either as an array or a red-black tree. Results are represented in milliseconds and are the mean of ten test runs. The test runs did not exhibit large variations.

Chapter 17

E-mail application

This chapter describes the design of a distributed e-mail system built on top of the XML Store framework. The purposes of building such a system are:

- Evaluation of the applicability of the API. Is the proposed API useful for building actual systems? Does it have the necessary methods and is it easy to work with?

- Is the XML Store system suitable for building an application like the e-mail system?

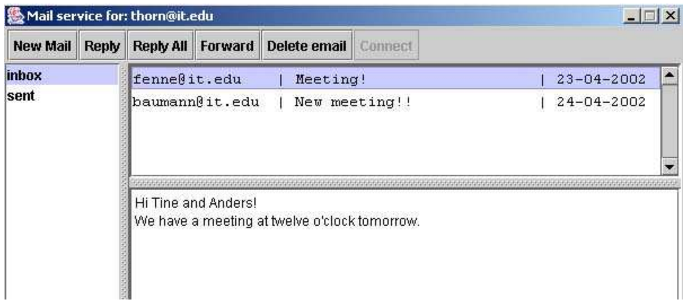
Notice that the purpose of the e-mail application is not to examine the general performance of the XML Store. This is done in chapter 16.

17.1 Description of the e-mail system

The e-mail application is kept very simple. It does not in any way implement the SMTP protocol, but simulates exchange of short messages using the XML Store framework. The main window of the application is shown in figure 17.1. The application has the following operations: Send, reply, reply all, forward and delete e-mail.

The basic behaviour of the application is very similar to existing e-mail applications such as *Netscape Messenger* and *Microsoft Outlook*. The left frame of the figure 17.1 is the folder frame, which contains the inbox and sent folder. The upper right folder is the e-mail folder and the lower left is the message folder. When the user selects a folder in the folder frame, the content of the folder is shown in the e-mail frame. When the user then selects an e-mail, the contents of the e-mail is shown in the message frame.

Before the user can use the e-mail system, she needs to press the *Connect* button, which makes the underlying XML Store try to join the XML Store network. Once the XML Store has successfully joined the network, the user is able to send and receive e-mails.

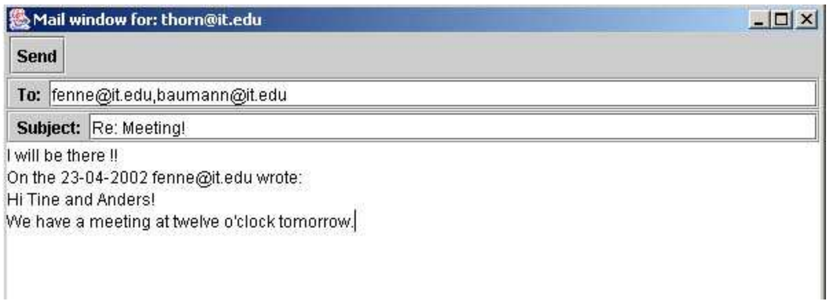


```
fenne@it.edu     | Meeting!               | 23-04-2002
baumann@it.edu   | New meeting!!          | 24-04-2002
```

Hi Tine and Anders!
We have a meeting at twelve o'clock tomorrow.

Figure 17.1: The main window of the e-mail application.

When the user wishes to send or reply to a e-mail, the e-mail application opens the e-mail window, which is shown in figure 17.2. In this window the user needs to specify the receiver, subject and message of the e-mail. Multiple recipients are separated by commas.



To: fenne@it.edu,baumann@it.edu
Subject: Re: Meeting!

I will be there !!
On the 23-04-2002 fenne@it.edu wrote:
Hi Tine and Anders!
We have a meeting at twelve o'clock tomorrow.

Figure 17.2: The e-mail window of the e-mail application.

Structure of e-mail folders

The folders `inbox` and `sent` are contained in a single XML document `mail_folders`, which is structured the following way:

```
<mail_folders>
   <inbox>
      <email>
         <date>
            23-04-2002
         </date>
         <to>
            <email_address>baumann@it.edu</email_address>
            <email_address>thorn@it.edu</email_address>
         </to>
         <sender>
            <email_address>fenne@it.edu</email_address>
         </sender>
         <subject>
```

```
            Meeting!
         </subject>
         <message>
            Hi Tine and Anders!
            We have a meeting at twelve o’clock tomorrow.
         </message>
      </email>
      <email>
         <date>
            24-04-2002
         </date>
         <to>
            <email_address>tine@it.edu</email_address>
            <email_address>fenne@it.edu</email_address>
         </to>
         <sender>
            <email_address>baumann@it.edu</email_address>
         </sender>
         <subject>
            New meeting!!
         </subject>
         <message>
            Hi Tine and Mikkel!
            We have a new meeting at eleven o’clock Friday.
         </message>
      </email>
   </inbox>
   <sent>
      <email>
         <date>
            23-04-2002
         </date>
         <to>
            <email_address>baumann@it.edu</email_address>
            <email_address>fenne@it.edu</email_address>
         </to>
         <sender>
            <email_address>thorn@it.edu</email_address>
         </sender>
         <subject>
            Re: Meeting!
         </subject>
         <message>
            I will be there!!

            On the 23-04-2002 fenne@it.edu wrote:
            Hi Tine and Anders!
            We have a meeting at twelve o’clock tomorrow.
         </message>
      </email>
   </sent>
</mail_folders>
```

Delivery of e-mails

The e-mail application makes use of the name service when an e-mail is delivered. The name service has an entry for each address (e.g. “thorn@it.edu”) in the system and each address points to the `mail_folders` document of the owner of the address. Sending an e-mail is accomplished by the following steps (the steps are illustrated in figure 17.3). The numbers in the figure corresponds to the steps below.

1. The name service is used to look up the value reference of the recipient's `mail_folders` document.
2. The XML Store is used to load the `mail_folders` document.
3. The e-mail is appended to the recipient's `inbox`.
4. The new `mail_folders` document is saved in the XML Store.
5. The name service is updated, so the receiver's address now points to the updated `mail_folders` document.

In a similar way, the e-mail is inserted into the `sent` folder of the sender.

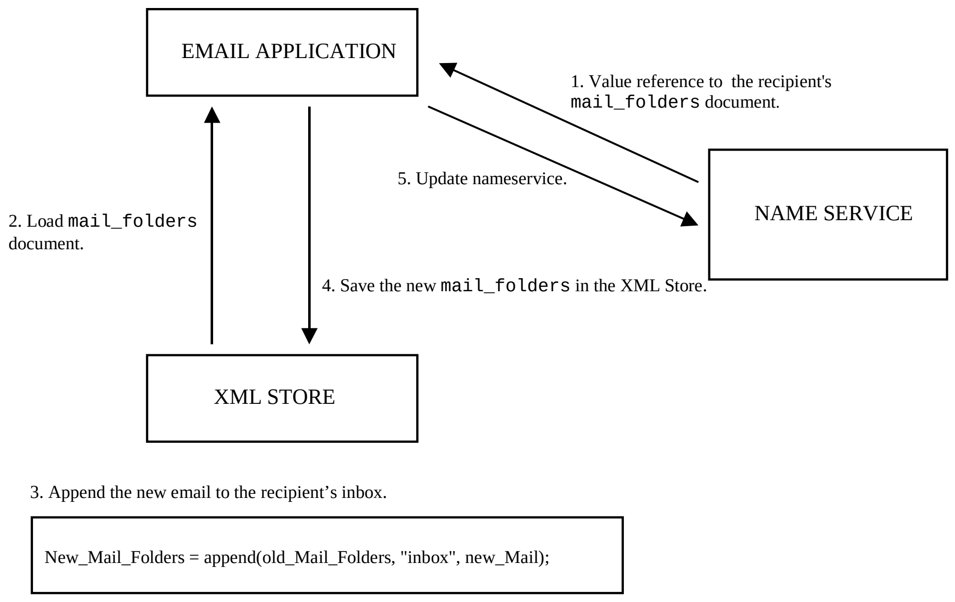

EMAIL APPLICATION
1. Value reference to the recipient's mail_folders document.
5. Update nameservice.
NAME SERVICE
2. Load mail_folders document.
4. Save the new mail_folders in the XML Store.
XML STORE
3. Append the new email to the recipient's inbox.
New_Mail_Folders = append(old_Mail_Folders, "inbox", new_Mail);

Figure 17.3: Delivery of an e-mail.

To keep the system as simple as possible, we have not implemented any security. Furthermore, we have chosen to ignore features like attachments, address book, unread messages and search facilities. These are all features, that could quite easily be implemented. For instance, unread messages can be implemented by maintaining a persistent list of value references of all the unread e-mails, and by removing a value reference from the list when the e-mail it refers to is read by the user.

17.2 Evaluation of API

During the implementation of the mail application we did not find any serious errors or deficiencies in the proposed API. We felt we had the necessary operations at our disposal, and the API was on the whole easy to work with. However, we did find some flaws and possible improvements.

Accessing the contents of a node is a frequently needed operation when working with XML documents. We found that this operation was rather awkward designed. For instance, to extract the actual e-mail address from an `email_address` element we have to write:

```
String address = emailAddress.getChildNodes().get(0).getValue();
```

To avoid this rather awkward series of method calls we propose a new method `getChildCharData(int index)` which is a shortcut for the above code. Using the new method we could write:

```
String address = emailAddress.getChildCharData(0);
```

Note that in the method an exception should be thrown if the child node at position `index` is not a character data node.

Another awkward feature of the API is the creation of documents. We propose a new method that takes a string with XML syntax as input, parses the input string and creates the relevant nodes. Instead of writing:

```
Node ch1 = createCharData("fenne@it-c.dk");
Node email = createElement("email_address", new node[] {ch1});
Node sender = createElement("sender", new node[] {email});
```

We could then write:

```
Node personae = createDocument("<sender><email_address>" +
 "fenne@it-c.dk</email_address></sender>");
```

The question of type safety was another issue we came across when developing the mail application. For instance, we have no way of knowing if the `<sender>` element actually contains an `<email_address>` element or if the `<email_address>` element contains an e-mail address. Therefore, an obvious improvement of the mail system would be to use a an XML Schema, which supports specification of data types, to validate the incoming XML documents. A more radical change to the XML Store would be a reflexive framework, that is able to automatically generate a class structure based on the element tags of an XML document. Java Architecture for XML Binding (JAXB) is an example of such a framework [68]. JAXB compiles an XML schema into one or more Java classes and the generated classes handle all the details of XML parsing and formatting. Similarly, the generated classes ensure that the constraints expressed in the schema are enforced in the resulting methods and Java data types. When using an architecture like JAXB our class structure of

`Node`, `Element` and `CharData` is thus replaced by a class structure custom-made to the `mailfolders` XML document. The `mail_folders` document is then represented by an instance of the `MailFolders` class which has two instances, `inbox` and `sent` of the `Folder` class, each containing a list of instances of the `Email` class etc. With this class structure we can easily retrieve the contents of an e-mail by accessing the fields of the e-mail object:

```
String email = email.sender.emailAddress;
```

The advantages of the described approach comes at the price of having to conform to a schema, that even has to be known at compile time. When you have to describe a schema you lose the generality and flexibility of semistructured data, and cannot handle any documents that do not conform to this schema.

17.3 Evaluation of suitability

Even though the e-mail application is very simple, the design and implementation still shows that the properties of the XML Store make it suitable for being the underlying framework of a distributed system like an e-mail application. The advantages of using the XML Store for this kind of application is high scalability as well as reduction of disk storage, which is due to sharing of subdocuments. Consider an example, where a user forwards an e-mail to 200 users. An ordinary e-mail server will produce 200 copies of the e-mail and put it in each of the 200 users' mailboxes. Some of the users will then forward the e-mail, other will just store it in a folder and yet others will never even read it. The result is several copies of the exact same e-mail. This redundancy is avoided in the XML Store, because the XML Store will only save the e-mail once and all users will then get a reference to this single e-mail.

Obviously, security is an issue we need to consider if the mail application is to be used in a realistic environment. As described in chapter 12 we have come up with a solution that makes it possible to encrypt values and still maintain sharing. Thereby, a user is prevented from reading another user's e-mail. However, the possibility of a malicious user deleting another user's e-mails is still a security problem, since the sender of an e-mail has access to the entire `mail_folders` document of the receiver. This problem can be solved if we customise the name service, so an address in the name service no longer contains a value reference to the `mail_folders` document of the owner of the address, but instead contains a value reference to a temporary inbox of the owner of the address. This way, the sender of an e-mail has access only to the temporary inbox, and hence cannot modify the real inbox of the receiver. When a user wants to read her inbox, she uses the name service to lookup the address of the inbox. The name service will then authenticate the user and append the e-mails in the temporary inbox to the real inbox. If the e-mail application should be realistic, then we also need to modify and optimise the design of the name service, so that it no longer constitutes a single point-of-failure.

To summarise, the design and implementation of the mail application has shown that the proposed API is applicable and easy to use, though it has some awkward methods. It has also shown that the XML Store is suitable for being the underlying framework of applications like the distributed mail system.

Chapter 18

Conclusion

In this thesis, we have presented the XML Store system: A self-organising, highly scalable distributed system for storing XML documents in a value-oriented manner, based on the Chord routing and location scheme.

Distributed storage The XML Store is based on the Chord routing and location protocol. We use the Chord protocol as a service that translates a location-independent value reference to a network route by which the value can be found.

By building on the Chord protocol, we show that it is possible to design and implement an efficient, value-oriented XML Store which is decentralised, self-organising, scalable, fault tolerant and provides high availability of data. We also give a simple, working implementation of the Chord protocol in Java, clarifying the workings of the various pseudo-code fragments presented in the articles on Chord.

We propose an extension to the Chord protocol, inspired by Clarke et al. [17], that allows the detection of collisions between the cryptographic content hashes used to identify data.

Conventional encryption prevents sharing of data, as each user's data is encrypted with the user's private key. We present the *Content Hash Encryption* technique inspired by the *convergent encryption* technique [67]. This technique secures data from being read by unauthorised users, while still preserving sharing of identical data between different users.

In the prototype implementation of the XML Store network communication is implemented by a synchronous request-reply protocol, where each remote method invocation is handled by a separate thread. This solution is obviously insufficient, and a asynchronous request-reply protocol with fault tolerance should be introduced to the system.

API We present a value-oriented API for processing XML documents. The API represents the XML document as an object structure, resembling DOM, except that all aspects of the API are value-oriented – any modifications to a document will result in a new document being built, leaving the former document unaltered.

The API is arguably as applicable and versatile as DOM, and gives the application programmer a full range of operations for reading and modifying XML documents. The versatility of the API is demonstrated by implementing an e-mail system, modelling the user's folders and messages as XML documents.

We demonstrate how the core operations of the API are implemented efficiently, making processing of large XML documents feasible. By representing child lists as balanced trees, we obtain an $O(\log N)$ run-time for accessing children and modifying the list.

The API still needs some derived utility operations and short cuts, making common tasks easy for the application programmer. The core operations, however, are sufficient.

Processing XML documents When storing documents in the XML Store, the documents are split up according to their inherent tree structure, and each node is stored separately. Since we work value-oriented, the storage strategy makes sharing of identical subdocuments possible.

Experiments show that due to the extensive sharing of subdocuments, modification of stored documents is extremely efficient compared to flat text files. Only the changed parts of a document need to be stored – not the entire document as in the conventional serialisation of XML documents. The tree structured storage format furthermore allows us to access parts of the document without having to load and parse the entire document.

However, the storage strategy chosen in the XML Store is far from optimal. Our experiments show that the number of nodes of an XML document is of greater importance when storing a document than the size of the nodes. Documents stored using our storage strategy, are typically split up in too many small parts, each requiring expensive network operations. We propose an alternate storage strategy that generates fewer and larger blocks, presumably making storage of documents faster, but also reducing the amount of sharing. The performance of the alternate storage strategy has yet to be determined.

It seems plausible that the XML Store has potential for being very efficient with a different storage strategy and network communication implementation, since CFS, which is also based on Chord, achieves speeds competitive with FTP.

We have shown how a peer-to-peer system designed for storing XML documents can be efficiently constructed to support dynamic networks by using the Chord protocol for routing and location of peers.

Due to our value-oriented approach, we are able to present some simple solutions to usually complex problems regarding distributed systems, such as transaction handling, caching- and replication management.

We present an approach to processing and storing XML documents that in many ways is superior compared to traditional technologies such as SAX and DOM.

The techniques presented in the thesis have the potential to enable many interesting developments in the area of distributed systems as well as XML processing.

Chapter 19

Future work

This section will describe some of the aspects of the XML Store that were beyond the scope of this project, but will be important to consider in future work with the XML Store project. The present work may be improved and extended in various ways:

Name service The name service presented in the XML Store is clearly inadequate. Efforts should be concentrated in two areas: First of all, by extending the functionality of the name service in various ways, such as adding more advanced management of names. Secondly, by making the name service distributed, so it no longer constitutes a single point of failure. Whether the Chord protocol is useful for providing a decentralised name service has yet to be explored.

Distributed garbage collection The XML Store system only allows insertion of data – never deletion. This of course means that the store will gradually be filled up, possibly with data that is no longer in use. Garbage collection of some sort is required, so values no longer in use can be removed. This is somewhat complicated in a distributed environment, but it has been done before. The CFS system for instance [2], adopts the simple approach of using leases to keep data alive. If a lease has not been renewed for a given period of time, the data is considered unused, and is deleted. More advanced distributed garbage collection schemes require research though.

Network communication Previous work by Dabek [2] has made it clear that an effective flow control protocol for XML Store traffic is necessary. UDP transport over the wide area network saves resources and avoids the expense of setting up connections, but instead sacrifices the flow control properties that TCP provides. More effort should also be put into making the network communication in the XML Store asynchronous.

Security A complete defense against malicious nodes has yet to be devised. Stoica et al. [43] are working on making the Chord protocol more secure to malicious attacks.

Search facility The XML Store system could benefit from a search facility, that allows a user to retrieve a list of all documents containing one or more keywords specified by the user or to perform database style queries.

One way to provide the “keyword” search would be to adopt an existing centralised search engine. A more ambitious approach would be to store the required index files using the XML Store itself. Stoica et al. [43] are currently working on developing a keyword search facility for the CFS system.

XPath and XQuery are W3C standards for expressing database style queries. With XPath, a set of nodes can be selected based on a regular path expression. However, this operation can be very expensive to perform if we are dealing with a large XML document and if the entire XML tree has to be scanned to retrieve the relevant nodes. To reduce the portion of data that has to be scanned, index schemes have been proposed. Current research on indexes for semi-structured data includes the projects on Dataguides [69], T-indexes [70] and TOXIN [26]. However, more research in the area of distributed indexes is still needed, especially regarding whether advantages can be gained from the value oriented approach: can we exploit that data is never updated when building an index, and when performing updates, can an index be generated incrementally from the index of a previous document?

Bibliography

[1] F. Dabek, M. F. Kaashoek, D. Karger, R. Morris, and I. Stoica. Wide-area cooperative storage with CFS. *SOSP '01*, October 2001.

[2] F. Dabek. A cooperative file system. Master's thesis, Massachusetts Institute of Technology (MIT), September 2001.

[3] A. Rowstron and P. Druschel. Storage management and caching in PAST, a large-scale persistent peer-to-peer storage utility. In *Proc. ACM SOSP '01*, 2001.

[4] P. Druschel and A. Rowstron. PAST: A large-scale persistent peer-to-peer storage utility. In *Proc. HOTOS Conf.*, 2001.

[5] G. Coulouris, J. Dollimore, and T. Kindberg. *Distributed Systems - Concepts and Design*. Addison-Wesley, 3rd edition, 2001.

[6] S. Abiteboul. Querying semi-structured data. In *ICDT*, pages 1–18, 1997.

[7] D. Suciu. An overview of semistructured data. *SIGACT News*, 29(4):28–38, December 1998.

[8] L. C. Paulson. *ML for the working programmer*. Cambridge University Press, 2nd. edition, 1996.

[9] F. Henglein. Desiderata for an applicative persistent store manageer (XML Store). *memo*, 2001.

[10] Jie Wu. *Distributed System Design*. CRC Press, 1st edition, 1999.

[11] David L. Mills. Simple network time protocol (sntp) version 4 for ipv4, ipv6 and osi. http://www.ietf.org/rfc/rfc2030.txt?number=2030.

[12] J. M. Crichlow. *The essence of distributed systems*. Prentice-Hall, 2000.

[13] R. Elmasri and S. B. Navathe. *Fundamentals of database systems*. Addison-Wesley, third edition, 2000.

[14] D. Bricklin. Thoughts on peer-to-peer. http://www.bricklin.com/p2p.htm, August 2000.

[15] N. Minar. Distributed systems topologies: Part 1. http://www.openp2p.com/lpt/a//p2p/2001/12/14/topologies_one.html, December 2001.

[16] N. Krishnan. The JXTA solution to P2P. http://www.javaworld.com/javaworld/jw-10-2001/jw-jxta_p.html, October 2001.

[17] 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.

[18] The Gnutella protocol specification v0.4. http://www.gnutella.co.uk/library/-pdf/gnutella_protocol_0.4.pdf.

[19] N. Minar. Distributed systems topologies: Part 2. http://www.openp2p.com/lpt/a//p2p/2002/01/08/p2p_topologies_pt2.html, August 2002.

[20] M. Doernhoefer. A technology without peer(-to-peer). http://www.acm.org/sigsoft/SEN/surf2604.html, July 2001.

[21] F. Dabek, E. Brunskill, M. F. Kaashoek, D. Karger, R. Morris, I. Stoica, and H. Balakrishan. Building peer-to-peer systems with Chord, a distributed lookup service. In *Proceedings of the 8th Workshop on Hot Topics in Operating Systems (HotOS-VIII)*, pages 81–86, 2001.

[22] A. Rowstron and P. Druschel. Pastry: Scalable, decentralized object location and routing for large-scale peer-to-peer systems. In *Proc. IFIP/ACM Middleware 2001*, 2001.

[23] B. McLaughlin. *Java and XML*. O'Reilly, 2000.

[24] P. Buneman. Semistructured data. In *Proceedings of the 16th ACM Symposium on Principles of Database Systems*, pages 117–121, 1997.

[25] S. Abiteboul, P. Buneman, and D. Suciu. *Data on the web - from relatons to semistructured data and XML*. Morgan Kaufmann, 2000.

[26] F. Rizzolo. ToXin: An indexing scheme for XML data. Master's thesis, Univerity of Toronto, 2001.

[27] E. R. Harold and W. S. Means. *XML in a nutshell: A desktop quick reference*. O'Reilly, 2001.

[28] D. Suciu. Semistructured data and XML. In *In Proceedings of the Seventh Conference on Knowledge Management*, 1998.

[29] http://www.xml.com/pub/a/1999/11/sml/index.html, 1999.

[30] http://www.brics.dk/ amoeller/xml/xml/sgml2sml.html.

[31] D. Park. http://www.docuverse.com/smldev/minxmlspec.html, November 2000.

[32] R. Hoque. *XML for real programmers*. Morgan Kaufmann, 2000.

[33] J. Clark and S. DeRose. XML path language (XPath). http://www.w3.org/TR/xpath, November 1999.

[34] J. Clark. XSL transformations (XSLT). http://www.w3.org/TR/xslt, November 1999.

[35] S. DeRose, E. Maler, and R. Daniel Jr. XML Pointer Language (XPointer) version 1.0 (work in progress). http://www.w3.org/TR/xptr, January 2001.

[36] Napster. http://www.napster.com/.

[37] Napster protocol specification. http://opennap.sourceforge.net/napster.txt, April 2000.

[38] D. Bindel, Y. Chen, P. Eaton, D. Geels, R. Gummadi, S. Rhea, H. Weatherspoon, W. Weimer, W. Wells, B. Zhao, and J. Kubiatowicz. Oceanstore: An extremely wide-area storage system. Technical Report UCB/CSD-00-1102, Computer Science Division, U. C. Berkeley, May 1999.

[39] J. Kubiatowicz, D. Bindel, Y. Chen, S. Czerwinski, P. Eaton, D. Geels, R. Gummadi, S. Rhea, H. Weatherspoon, W. Weimer, C. Wells, and B. Zhao. OceanStore: An architecture for global-scale persistent storage. In *Proceedings of ASPLOS 2000*, November 2000.

[40] M. Ripeanu and I. Foster. Mapping the Gnutella network: Macroscopic properties of large-scale peer-to-peer systems. Electronic Proceedings for the 1st International Workshop on Peer-to-Peer Systems (IPTPS ’02), March 2002.

[41] H. Zhang, A. Goel, and R. Govindan. Using the small-world model to improve Freenet performance. In *Proceedings of IEEE, Infocom*, 2002.

[42] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishan. Chord: A scalable peer-to-peer lookup service for internet applications. *ACM SIGCOMM, San Deigo, CA*, 2001.

[43] I. Stoica, R. Morris, D. Liben-Nowell, D. Karger, M. F. Kaashoek, F. Dabek, and H. Balakrishan. Chord: A scalable peer-to-peer lookup service for internet applications. LCS Tech Report, Massachusetts Institute of Technology (MIT), January 2002.

[44] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishan. Chord: A scalable peer-to-peer lookup service for internet applications. Unpublished.

[45] C. G. Plaxton, R. Rajaraman, and A. W. Richa. Accessing nearby copies of replicated objects in a distributed environment. In *ACM Symposium on Parallel Algorithms and Architectures*, pages 311–320, 1997.

[46] S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Shenker. A scalable content-addressable network. In *Proc. ACM SIGCOMM '01*, 2001.

[47] B. Zhao, J. Kubiatowicz, and A. Joseph. Tapestry: An infrastructure for fault-tolerant wide-area location and routing. Technical Report UCB/CSD-01-1141, Computer Science Division, U. C. Berkeley, April 2001.

[48] K. Hildrum, S. Kubiatowicz, J. Rao, and B. Zhao. Distributed data location in a dynamic environment. Technical Report UCB/CSD-02-1178, Computer Science Division, U. C. Berkeley, April 2002.

[49] OpenNap: Open source Napster server. http://opennap.sourceforge.net/.

[50] Software AG. Tamino XML Server - White paper. http://www.softwareag.com/tamino/download/tamino.pdf, 11 2001.

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

[52] Li Gong. Project JXTA: A technology overview. http://www.jxta.org/project/www/white-papers.html, 2002.

[53] A. Le Hors and P. Le Hegaret. Document object model (dom) level 2 core specification. http://www.w3.org/TR/2000/REC-DOM-Level-2-Core-20001113/, November 2000.

[54] T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to algorithms*. MIT Press, 1st edition, 1990.

[55] C. Okasaki. Functional pearls: Red-black trees in a functional setting. *J. Functional Programming*, 9(4):471–477, July 1999.

[56] S. Kahrs. Red-black trees in a functional setting. http://www.cs.ukc.ac.uk/people/staff/smk/, 2000.

[57] D. Karger, E. Lehman, T. Leighton, M. Levine, D. M. Lewin, and R. Panigrahy. Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the World Wide Web. In *Proceedings of the 29th Annual ACM Symposium on Theory of Computing*, pages 654–663, 1997.

[58] D. M. Lewin. Consistent hashing and random trees: Algorithms for caching in distributed networks. Master's thesis, Massachusetts Institute of Technology (MIT), May 1998.

[59] http://wombat.doc.ic.ac.uk/foldoc/foldoc.cgi?cache.

[60] K. Cheng and Y. Kambayashi. LRU-SP: A size-adjusted and popularity-aware LRU replacement algorithm for Web caching. In *Proceedings of the 24th IEEE Computer Society International Computer Software and Applications Conference (Compsac'2000)*, pages 48–53. IEEE Computer Society, October 2000.

[61] D. Lee, J. Choi, J.-H. Kim, S. H. Noh, S. L. Min, Y. Cho, and C. S. Kim. LRFU (least recently/frequently used) replacement policy: A spectrum of block replacement policies. citeseer.nj.nec.com/article/lee96lrfu.html, 1996.

[62] J. Gwertzman and M. I. Seltzer. World Wide Web cache consistency. In *USENIX Annual Technical Conference*, pages 141–152, 1996.

[63] A. J. Menezes, P. C. van Oorschot, and S. A. Vanstone. *Handbook of applied chryptography*. CRC Press, 1996.

[64] M. Rosenblum and J. K. Ousterhout. The design and implementation of a log-structured file system. *ACM Transactions on Computer Systems*, 10(1):26–52, 1992.

[65] B. R. Preiss. *Data structures and algorithms with object-oriented design patterns in Java*. John Wiley and Sons, Inc., 2000.

[66] J. T. Pedersen and K. B. Pedersen. A distributed XML Store. Master's thesis, IT-University, Copenhagen, August 2002.

[67] W. J. Bolosky, J. R. Douceur, D. Ely, and M. Theimer. Feasibility of a serverless distributed file system deployed on an existing set of desktop PCs. In *Proceedings of the international conference on measurement and modeling of computer systems*, pages 34–43, 2000.

[68] JavaTM architecture for XML binding (JAXB). http://java.sun.com/xml/jaxb/.

[69] R. Goldman and J. Widom. DataGuides: Enabling query formulation and optimization in semistructured databases. In *Proceedings of the 23th VLDB Conference*, 1997.

[70] T. Milo and D. Suciu. Index structures for path expressions. *Lecture Notes in Computer Science*, 1540:277–295, 1999.