A peer-to-peer, value-oriented XML Store

Tine Thorn, Anders Baumann, Mikkel Fennestad and Peter Sestoft
IT University of Copenhagen, Denmark
{*thorn, baumann, fenne, sestoft*}*@it.edu*

2003-05-19

Abstract

This paper presents and evaluates XML Store, a distributed and highly scalable storage system for XML documents that builds on the Chord peer-to-peer protocol developed at MIT. The interface (API) to the storage system is simple and value-oriented, permitting no destructive updates, which permits simple solutions to replication, caching, and transaction management.

Our experimental implementation of XML Store and the Chord protocol in Java performs satisfactorily and demonstrates the advantages of value-oriented programming.

1 Introduction

The growth in storage, bandwidth, and computational resources has fundamentally changed the way that applications are constructed, and has inspired a new class of distributed peer-to-peer storage infrastructures. Existing peer-to-peer systems such as CFS [7, 5], PAST [17, 8] and OceanStore [3, 12] seek to take advantage of the rapid growth of resources to provide inexpensive, highly available storage without centralised servers. These systems are based on protocols (Chord [6, 18], Pastry [16] and Tapestry [24]) that map data to servers, thereby essentially providing distributed hash table functionality.

We present the XML Store, a distributed storage infrastructure for storing XML documents. Henglein [10] provided the initial idea of a value-oriented XML Store, which is part of the ongoing "Plan 10" project. Like the above peer-to-peer systems, the XML Store relies on a peer-to-peer distributed hash table, namely Chord [6, 18] that supports storage and retrieval of data as sequences of bytes associated with a 160-bit hash value. However, the XML Store does not offer a conventional file system interface, but rather transparent storage of XML documents. The XML Store furthermore provides a rich API for processing and manipulating XML documents.

In this paper we focus on the design of the XML Store and the value-oriented approach. The central idea of value-oriented programming is that data are immutable, and that computation proceeds by creating new values, not by altering existing data as in the prevalent approach known from imperative programming languages. In such programming languages, data are modified destructively and consequently the original data are lost. This precludes sharing and complicates replication. Destructive updates of data cause problems in a distributed environment making cache management and transactions difficult.

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 destructive updates. Cache management and consistency protocols are unnecessary, and updates can be performed atomically without complex transaction control.

The XML Store system stores XML documents using the value-oriented approach. Although XML is quite widespread today, most XML document handling uses traditional methods, storing XML documents in text files in a conventional file system. By using value-oriented techniques and storing XML documents according to their inherent tree structure, possibilities arise for more elegant XML processing. For instance, sharing of identical subdocuments and lazy loading become possible.

The focal point of this paper is the value-oriented approach to storing XML combined with a peer-to-peer distributed hash table. Other interesting aspects of distributed storage systems such as privacy, anonymity, security, being tamper proof and censorship resistant are interesting, but not a goal in this paper.

The remainder of this paper is organised as follows. Section 2 discusses related work. Section 3 introduces value-oriented programming. Section 4 outlines the design of the XML Store and briefly describes Chord, used for routing and location in the XML Store. Sections 5 and 6 describe the properties and implementation of the XML Store, respectively. An evaluation of the XML Store is presented in section 7. Section 8 summarises our findings and conclude the paper. Finally, section 9 discusses future work.

2 Related work

The XML Store was inspired by CFS [6, 7] and by various technologies for storing and processing XML documents. This section presents related technologies for storing XML documents as well as related peer-to-peer storage systems.

2.1 Text files

Most XML documents are stored as text files in a conventional file system. Document processing requires parsing, possibly using frameworks such as SAX and DOM. This approach has some drawbacks. First of all, only serial access is possible. To build an abstract representation of a document it is necessary to parse the entire file into an in-memory representation. This limits the size of XML documents that can be handled and is inefficient when only parts of the document need to be accessed. Conventional file systems are furthermore not well suited for managing concurrent updates. In a multi-user environment these issues must be handled by the application programmer.

The main advantage of storing XML documents in text files is that no conversion is necessary when exchanging information, as the data are already available in the format that applications expect.

2.2 Relational databases

Another approach is to store XML documents in a relational database. The main advantage is that the database handles issues such as concurrency control, data integrity and scalability in a multi-user environment.

Using a relational database to store XML documents requires complex mappings of XML data to database entities. It is possible to create a database schema to store all kinds of XML documents, but this approach is inefficient since the data model of XML is very different from the relational data model [21], and leads to very complex and inefficient SQL queries.

Typically the application programmer will require a DTD or XML Schema, prior to creation of the database tables.

2.3 Native XML databases

Native XML databases are designed for storing and processing XML data. Data are inserted as XML and retrieved as XML.

Tamino XML Server [1] and Apache Xindice [2] are both native XML databases. 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. Unlike our XML Store, which is based on a peer-to-peer system, Tamino and Xindice are centralised server solutions. They 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.

2.4 Peer-to-peer systems

The popularity of peer-to-peer file sharing systems such as Napster [23] and Gnutella [22] has created a flurry of recent research activity into peer-to-peer architectures such as CFS [7, 5], PAST [17, 8] and OceanStore [3, 12].

Although the exact definition of a “peer-to-peer system” is debatable, such a system typically does not have dedicated, centralised infrastructure, but rather depends on the voluntary participation of peers to contribute resources out of which the infrastructure is constructed.

In Napster, 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 name and obtains the location (the IP address of a user machine) that stores the requested file. The process of locating a file is thus very much centralised and therefore, Napster is not a “pure” peer-to-peer system.

Gnutella is a decentralised peer-to-peer system and has no central servers to query. Instead, peers in the Gnutella network use broadcast to locate files. Peer-to-peer systems such as XML Store, CFS [7, 5], PAST [17, 8] and OceanStore [3, 12] seek to provide highly

	XML Store	CFS	PAST	Ocean-Store	Napster	Gnutella
Routing & location	Chord	Chord	Pastry	Tapestry	Central server	Broad-cast
Deterministic location	Yes	Yes	Yes	Yes	No	No
Scalable	Yes	Yes	Yes	Yes	No	Limited
Decentralised	Yes	Yes	Yes	Yes	No	Yes
Split up documents	Yes	Yes	No	No	No	No
Immutable data	Yes	Yes	Yes	No	No	No

Table 1: Features and properties of related peer-to-peer systems.

available storage with efficient location-independent routing and location in a decentralised environment.

These peer-to-peer systems rely on novel distributed algorithms for routing and locating. XML Store and CFS are based on Chord [6, 18], PAST is based on Pastry [16], and OceanStore is based on Tapestry [24].

All these algorithms resemble distributed hash tables, supporting the basic operations: insert and lookup of $\langle$key,value$\rangle$ pairs, where key = hash(value) and the hash function provides a deterministic, random, uniform distribution. In essence, the algorithms distribute $\langle$key,value$\rangle$ pairs across various peers in a large network, in a manner that facilitates scalable access to data using the key.

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 bottlenecks.

Napster [23] is not considered scalable, since it relies on a central server to locate files and this results in a performance bottleneck. Gnutella [22] displays a limited scalability because of the extensive use of broadcasting. The broadcast based protocol incurs high bandwidth requirements.

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

Table 1 gives an overview of the most important properties and features of the discussed peer-to-peer systems CFS, PAST, OceanStore, Napster, Gnutella and the XML Store. A more detailed survey of related work in the field of peer-to-peer storage systems can be found in [20].

3 Value-oriented programming

Value-oriented programming is programming with values and value references, which are themselves values. Values are by definition immutable. This principle apply not only 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 conventional programmer, seem alien and inconvenient. As the imperative paradigm revolves around assignment and hence modification of data, the inability to destructively update may seem like a limitation in the programming model. As is known from functional programming this is not the case, provided that new values can be created easily.

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 back to source.

Value-oriented programming adopts a “share-and-create” style known from functional programming languages, such as Standard ML, Scheme and Haskell. Value-oriented programming is characterised by sharing as much data as possible and only copying data if there is a need for update [10]. The three steps from above correspond to:

1. Obtain 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.

Value-oriented concepts are used not only in functional programming languages. For instance, Java's strings are immutable, which means that every modification must create a new `String` object, leaving the original object unchanged. This permits sharing and efficient comparison and parameter passing of strings.

3.1 Value-oriented trees

We focus on tree-structured data, because XML is a linear syntax for describing labelled trees. Consider the two trees in figure 1, which represent two XML documents. Instead of having two isomorphic subtrees, the trees have a reference to the common subtree. Note that a reference to a subtree is not a conventional pointer, but rather a *value reference*. A value reference is an identifier for a value and can be used to retrieve a value by using a value reference resolver. When replacing one of the subtrees in figure 1 by a value reference, 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 both of the original two trees by traversing the dag from the two root nodes. Observe that sharing is only possible because value-oriented programming provides a semantic guarantee: *No updates to substructures*.

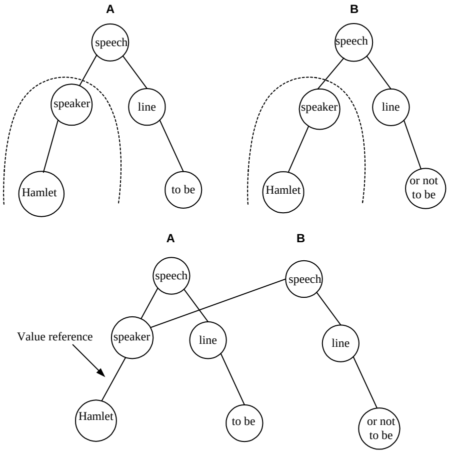


Figure 1: Isomorphic subtree shared by two trees A and B.

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 (modified) tree, sharing unchanged parts. Figure 2 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 may seem since A' shares most of its nodes with A and at most $depth(A)$ new nodes are introduced in A'.

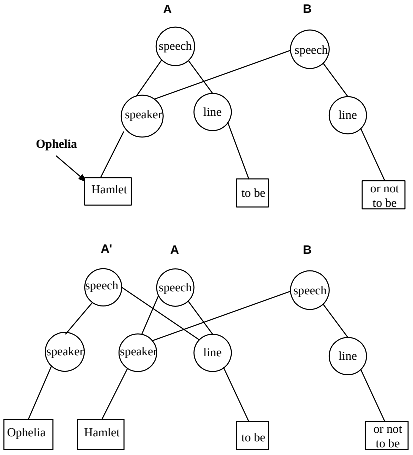


Figure 2: Modification of a tree using value-oriented programming.

Distributed systems need *caching* to provide fast access to shared data, and *replication* to increase fault tolerance. But caching and replication introduce problems with inconsistency if data can be updated. Immutable data have the advantage that they require no cache coherence management. Furthermore, immutable data can be freely replicated and coalesced (de-replicated and shared) without the need of complex replication protocols.

4 Design overview

Figure 3 shows the XML Store, the Name Service and an application using these two components. The XML Store is the main component, providing an API for processing and manipulating XML documents as well as making storage and retrieval of these documents possible.

When storing a given document, the XML Store re-

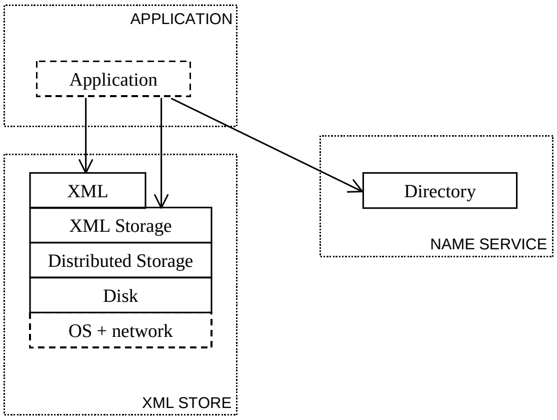


Figure 3: An application using the XML Store system, where four layers are shown.

turns 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*, which maps symbolic names to value references.

The purpose of the Name Service is to build useful distributed applications, because we need to be able to associate XML documents with names readable by humans. Clients cannot share particular resources managed by a computer system unless they can name them consistently. Thus, names facilitate communication and resource sharing [4]. The entire world is not value oriented: Exchange rates change, newspapers publish new articles, the weather forecast changes, etc. Therefore it must be possible to update the associations between name and document. Other applications interested in the document can obtain the value reference for a document from the Name Service. The Name Service is completely separated from the XML store, as the XML Store only knows of values and value references and not of any symbolic names. It may seem that we have simply moved the update problem (and the cache and replica consistency problems) to the Name Service. This is true, but updating the Name Service will be less frequent than updating data.

The XML Store consists of four layers, each responsible for a clearly defined set of tasks, as shown in table 2. In the remainder of the paper, a *peer* is a server in the distributed XML Store and a *node* is an XML node.

Layer	**Example Operations**
XML	Various methods for traversal and modification of XML documents, 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)`

Table 2: Operations of the layers in the XML Store component.

4.1 The Disk layer

Each individual peer in the system uses the Disk layer for permanent storage of byte sequences associated with their hash codes. The Disk layer thus has a hash table interface and simply stores and retrieves ⟨byte sequence, hash code⟩ pairs using the local file system.

4.2 The Distributed Storage layer

The Distributed Storage layer is a peer-to-peer system based on the Chord protocol [6, 18]. It can be regarded as a distributed hash table that supports storage and retrieval of sequences of bytes associated with a (hash) key.

The Distributed Storage layer is inspired by CFS [5], 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.

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, described above.

The current XML Store system only allows insertion of data – never deletion. This means that the store will always grow, possibly containing data that are no longer in use. Some sort of garbage collection is required, so that such values can be removed. This is complicated in a distributed environment. The CFS system [5] adopts the simple approach of using leases to keep data alive for a limited time only. If a lease for a data block expires the block is considered unused, and is deleted. Better distributed garbage collection schemes are required, but has not yet been implemented.

4.2.1 Chord

The Chord protocol developed at MIT [18] solves the central problem: Given a key, locate the peer responsible for that key and the associated value. Given this functionality it is easy to implement a distributed hash table supporting storage and retrieval of $\langle$key,value$\rangle$ pairs.

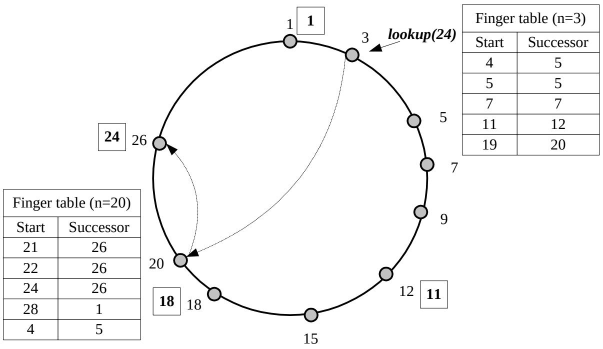
The Chord protocol takes as input an m-bit key and returns the id (IP number) of some peer on the Chord identifier circle that holds the corresponding key. The Chord system uses a variant of *consistent hashing* [11] to map keys to peers. In contrast to standard hashing techniques, different sets of cells do not induce completely different mappings of keys to cells. 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 peers to points on a circle using a base hash function [13]. In Chord this is accomplished by assigning each peer and each key a probabilistically unique m-bit identifier generated by a base hash function such as the cryptographic hash function SHA-1. An identifier for a peer is obtained by hashing a unique identifier of the peer such as its IP-address in an Internet environment.

In Chord a key is mapped to a peer id in the following way: A key, k, is mapped to the first peer whose identifier, id, is equal to or follows k in the identifier space (i.e. on the Chord identifier circle). The peer responsible for k is the *successor* of k's identifier [7].

To guarantee a correct lookup Chord only requires that the following invariant is maintained: Each peer must know its immediate successor. Performing lookups by stepping through the Chord identifier circle via successor pointers guarantees that the correct peer is returned, but this strategy is inefficient – in the worst case every peer needs to be traversed leading to a run time complexity of $O(N)$, where N is the total number of peers in the system.

To improve the run time of lookup operations, additional routing information is needed: Every peer 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. In an N peer Chord system, each peer maintains information about $O(\log N)$ other peers in the finger table. Using finger tables to perform a lookup requires $O(\log N)$ messages [19].



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 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. Peers 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 peer which they have been assigned. The finger tables of peer 3 and 20 are shown.

4.3 The XML Storage layer

The *XML Storage* layer implements the storage strategy. It is responsible for converting documents represented as an XML node structure in the XML layer to sequences of bytes that the underlying Distributed Storage layer can understand.

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 dag 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 XML node as a separate value. For the application programmer, an XML document as a whole is represented by the root node of the document. When the document is saved the tree 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 ⟨value reference, byte-representation⟩ pair is placed on the appropriate Chord peer located by the Distributed Storage layer using the Chord protocol (section 4.2). By letting a cryptographic hash value (SHA-1) act as value reference, the value reference is completely determined by the value alone, of fixed size, location independent and with high probability unique to that value (i.e. distinct values map to distinct value references).

For simplicity we use a string based format for representing the nodes as bytes, even though a binary format would be more space 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 5 illustrates this approach by showing an XML document and the values that it consists of when stored using the described strategy. The example uses symbolic value references ($r0 - r7$) for readability; in reality they are 160-bit binary numbers. The suggested storage strategy has two important properties that make modification and storing of large XML documents very efficient.

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 5: The storage strategy of the XML Store system. The document on the left is stored as the eight strings on the right.

First, because we work with immutable values, sharing of subdocuments is possible. When a document is loaded and part of it is modified we only have to save the parts of the document affected by the modification, namely the path from the root to the modification. The remaining nodes are already stored and there is no need for storing them again. If one for example wished to add information about the playwright of the play to the document in figure 5 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 saves a lot of storage space under the assumption that most XML documents are copies, slight modifications or aggregates of other XML documents.

Secondly, the storage strategy 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 5 can for example be accessed by traversing the `play/title` path, making it unnecessary to load the entire play which may include 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 know only their own value reference. When asked for content the proxy simply loads the actual node.

A save operation is expensive due to network overhead. To prevent saving a large number of very small nodes, nodes are inlined and only saved when they exceed a certain minimum size. Furthermore most of the time spent in a save operation is used for waiting during network communication – if we store nodes sequentially much time will be spent waiting for a save operation to be finished before the next can be initiated. To save time we therefore store a number of nodes concurrently.

4.4 The XML layer

The *XML* layer models XML documents as an object structure, allowing the programmer to work with an abstract tree 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 corresponds to a subset of the methods offered by DOM. However, the DOM API is imperative, which means that XML nodes in the document tree are updated destructively. In contrast we provide a value-oriented API, which, besides being both as 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. A full description of the value-oriented XML API can be found in [20], chapter 7.

5 Properties

The design of the XML Store has the following desirable properties:

Simple implementation: The value-oriented approach simplifies usually complex problems in distributed systems, such as transaction handling and cache and replication management. In addition, the value-oriented approach makes it easier to maintain several versions of a document.

Efficient XML processing: As a consequence of working value-oriented and storing XML documents according to their inherent tree structure, sharing and lazy loading are made possible. Thereby, the XML Store provides possibilities for elegant and efficient processing of XML documents.

Applicable API: The XML Store presents a simple and flexible value-oriented API for processing and manipulating XML documents. The API is easy to use and is as versatile as DOM.

Moreover, by building the XML Store on top of the Chord protocol we automatically achieve the following properties:

Decentralised: The XML Store becomes fully distributed – no peer holds more information than any other peer in the system.

Scalable: Lookup operations operate in time logarithmic in the number of peers and only requires logarithmic routing information at each peer. Therefore, the XML Store scales gracefully when the number of peers and amount of data increases.

Load balance: Chord uses consistent hashing to map values to peers, and thereby data are spread evenly around the Chord identifier space. In the XML Store system, XML documents are split up into fragments according to the inherent tree structure. XML documents are hence distributed across multiple peers, which increases load balancing, as popular files are spread across multiple peers.

Fault tolerance and availability: Being based on the Chord protocol it is easy to implement fault tolerance and availability of data by using successor lists and replication, respectively. This has been done in CFS and the techniques described there can be applied without modification.

Self-organising: With the above measures implemented the XML Store system is able to automatically adapt to the arrival, departure and failure of peers.

6 Implementation

We have implemented a prototype of the XML Store in Java 1.4. It is available from `www.it.edu/xmlstore/`. For a detailed description of the implementation see [20].

The prototype implements the Chord protocol but does not implement any measures for fault tolerance and availability of data. These features have been implemented and described by Dabek et al. [7] in the CFS system and can be added without alterations.

The current prototype has been subjected to a large number of experiments and is expected to be fairly bug free.

7 Results & evaluation

A number of experiments with the proof-of-concept prototype have been conducted to evaluate the performance and scalability of the XML Store and the usefulness of the programming model. The experiments show that the time taken to store a document is mostly dependent on the number of nodes in a document, and to a lesser degree the size of the nodes. This is understandable since each node is stored separately requiring fairly expensive network communication. The first prototype of the XML Store used TCP/IP for network communication, but this was too slow for sending short request-reply messages. The current XML Store uses a UDP-based protocol with error correction and flow control, and is considerably faster, but network communication is still a bottleneck.

The experiments furthermore reveal that the XML Store shows satisfactory performance when storing nodes. The prototype can store between 35 and 60 XML nodes per second with an average of approximately 38 nodes per second. Loading of files is a bit slower (31 nodes per second on the average) probably because nodes are not loaded concurrently. The experiment was conducted in a controlled test environment with 50 XML Store peers, configured to store 10 nodes concurrently and only inline subtrees smaller than the size of a value reference.

It is possible to improve the performance of the save and load operations by increasing the *inlining constant* – the size limit that determines whether a node is small enough to be inlined in its parent or whether it should be saved individually. By inlining more nodes we reduce the number of network operations, as fewer and larger blocks of data are stored, thereby increasing the system's overall performance. More inlining unfortunately also decreases sharing: Identical subtrees are still shared, but the amount of sharing is reduced, because identical nodes that previously would have been shared, can be inlined in different contexts. More research is needed to determine the actual impact on sharing.

The experiments demonstrate the benefits of the value-oriented approach when storing modifications of documents. Saving an XML document using conventional technologies, such as text files, will take time linear in the size of the document. If instead an XML document is saved using the described approach, the time it takes to save a document will be linear in the size of the *changes* made to the document. Table 3 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. The experiment was conducted in the same test setup. As can be seen from the results, it takes some time to save the XML document the first time. Saving the document again, after modifying it in various 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 previous versions of the document still exists in the XML Store.

Operation	**Inlining: 31**	**Inlining: 250**
Save "Hamlet"	105042 msec	4006 msec
Add character	129 msec	26 msec
Modify title	139 msec	41 msec
Add speech	182 msec	46 msec
Modify line	108 msec	41 msec
Delete (final) scene	142 msec	38 msec

Table 3: Results of saving an XML document (274 KB) repeatedly after modifying it in various ways. In the first experiment subtrees are inlined only if they are smaller than the size of a value reference (31 bytes) and in the second one when they are smaller than 250 bytes.

The versatility of the API has been demonstrated also by implementing a distributed e-mail application on top of the XML Store. The design and implementation of the e-mail system show that the proposed API is applicable and that the XML Store is suitable for being the underlying framework of applications like the distributed e-mail system [20]. However, larger and more realistic examples are needed to fully assess the usefulness of the programming model.

8 Conclusion

We have presented the design of the XML Store, a peer-to-peer system for storing XML documents. By building on the Chord protocol, the XML Store becomes scalable, decentralised, self-organising and provides some degree of load balancing.

When storing XML documents we employ a value-oriented storage strategy and split up XML documents according to their inherent tree structure. Each subdocument is considered a value and is 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 and preserve sharing of identical subdocuments. The tree structured storage format moreover makes lazy loading possible, i.e. parts of a document can be accessed without having to load and parse the entire document. The value-oriented approach furthermore provides simple solutions to usually complex problems in distributed systems, such as transaction handling, caching and replication and it makes it easier to maintain several versions of a document.

On top of the storage infrastructure we have built a value-oriented API for processing XML documents. The API represents XML documents 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 old 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 has been demonstrated by implementing an e-mail system, modelling the user's folders and messages as XML documents.

Our experiments show that due to the extensive sharing of subdocuments, modification of stored documents is extremely efficient compared to storage using 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 techniques presented in this paper have the potential to enable many interesting developments in the area of distributed systems as well as XML processing.

9 Future work

Directions for future work on the XML Store project includes security, garbage collection, and search facilities.

The current XML Store implementation does not include security features. However, in [20] we present 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. Furthermore, it is difficult to tamper with data that resides in the XML Store without the tampering being easily detectable. However, we still need to consider security measures for preventing vandalism.

A more advanced form of garbage collection than timed leases is required, so that values that are no longer in use can be removed.

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

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 document 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 [9], T-indexes [14] and TOXIN [15]. However, more research in the area of distributed indexes is needed, especially regarding whether advantages can be gained from the value oriented approach.

10 Acknowledgements

Fritz Henglein proposed the concept of a value-oriented XML Store and provided much inspiration and encouragement for this work.

References

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

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

[3] 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.

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

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

[6] 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.

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

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

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

[10] F. Henglein. Desiderata for an applicative persistent store manager (XML Store). *Memo, Department of Computer Science, University of Copenhagen*, 2001.

[11] 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.

[12] 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.

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

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

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

[16] 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.

[17] 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.

[18] 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.

[19] 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.

[20] T. Thorn, A. Baumann, and M. Fennestad. A distributed, value-oriented XML Store. Master's thesis, IT-University, Copenhagen, August 2002.

[21] F. Tian, D. DeWitt, J. Chen, and C. Zhang. The design and performance evaluation of alternative XML storage strategies, 2000.

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

[23] Napster website. Napster. http://www.napster.com/.

[24] 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.