Storm: Supporting data mobility through location-independent identifiers

Benja Fallenstein, Tuomas J. Lukka and Hermanni Hyytiälä
Dept. of Mathematical Information Technology
University of Jyvskyl, Finland
PO. Box 35 (Agora), FIN-40014 Jyvskyl
b.fallenstein@gmx.de, lukka@iki.fi, hemppah@cc.jyu.fi

Toni Alatalo
Department of Information Processing Sciences
University of Oulu, Finland
P.O.Box 3000, FIN-90100 Oulu
antont@oulu.fi

May 9, 2003

Abstract

In this paper, we define data mobility as a collective term for the movement of documents between computers, different locations on one computer and movement of content between documents. We identify dangling links and alternative versions as major obstacles for the free movement of data. This paper presents the Storm (STORage Module) design as one possible solution to these problems. Storm uses location-independent globally unique identifiers, append-and-delete-only storage and peer-to-peer networking to resolve problems raised by data mobility. Moreover, we discuss some specific use scenarios related to ad hoc networks, unreliable network connections and mobile computing, in which the need for data mobility is obvious. Our current prototype implementation works on a single system; peer-to-peer networking is in an early prototype stage. H.5.4Information Interfaces and PresentationHypertext/Hypermedia[architectures] H.3.4Information Storage and RetrievalSystems and Software[distributed systems, information networks]

Design, Reliability, Performance

versioned hypermedia, dangling links, xanalogical storage, peer-to-peer, location-independent identifiers

1 Introduction

The Web and several other hypermedia systems assume that identifiers either have to include location information (as in regular URLs, which break when documents are moved), or can only be resolved locally (as in link services that can only find links stored on a select set of link servers [21, 9]). Berners-Lee [4] argues that unique random identifiers are not globally feasible for this reason.

However, recent developments in peer-to-peer systems have rendered this assumption obsolete. Structured overlay networks [44, 38, 50, 41, 32, 31, 3, 5] allow location-independent identifiers to be resolved on a global scale. Thus, it is now possible to perform a global lookup to find all information related to a given identifier on any participating peer in the network. This, we believe, may be the most important result of peer-to-peer research with regard to hypermedia.

In this paper, we examine how location-independent identifiers can support *data mobility*. Documents often move quite freely between computers: they are sent as e-mail attachments, carried around on disks, published on the web, moved between desktop and laptop systems, downloaded for off-line reading or copied between computers in a LAN. We use 'data mobility' as a collective term for the movement of documents between computers (or folders!), and movement of content between documents (through copy&paste)[1].

We address two issues raised by data mobility: Dangling links and keeping track of alternative versions. Resolvable location-independent identifiers make these issues much easier to deal with, since data can be identified wherever it is moved[2]. Current systems dealing with these issues often do not deal well with many forms of data mobility.

Dangling links are an issue when documents are moved between servers; when no network connection is available, but there is a local copy (e.g. on a laptop or dialup system); or when the publisher removes a document permanently, but there are still copies (e.g. in a public archive such as [22]). Dangling links are also an issue when a document and a link to it are received independently, for example as attachments to independent emails, or when a link is sent by mail and the document is available from the local intranet. When two people meet e.g. on the train, they should be able to form an ad-hoc network and follow links to documents stored on either one's computer [46]. Furthermore, when a document is split to parts, links to the elements in the parts that are then in new documents should not break.

Advanced hypermedia systems such as Microcosm and Hyper-G address dangling links through a notification system [21, 24]: When a document is moved, a message is sent to servers storing links to it. Hyper-G uses an efficient protocol for delivering such notifications on the public

[1]While the physical mobility of e.g. notebooks may effect data mobility (for example due to caching for off-line access), data mobility is neither the same as, nor limited to the physical movement of devices.

[2]It might be more appropriate to speak about *resources* and *references* instead of *documents* and *links*, but in the spirit of [24], we stick with the simpler terms for explanation purposes.

Internet.

Location-independent identifiers for documents make such a system unnecessary; a structured peer-to-peer lookup system can find documents wherever they are moved. This kind of system also works for data not publicized on the Internet. For example, if one email has a document attached to it, and another email links to this document, an index of locally stored documents by permanent identifier allows the system to follow the link. This would be difficult to realize through a notification mechanism.

Tracking alternative versions, on the other hand, is an issue when documents are modified on several independent, unconnected systems, for example when a user takes a document home from work on a floppy disk; when they keep the same set of documents on their desktop and laptop, modifying them on each; when two people collaborate on a document, sending each other versions of the document by email; when someone downloads a document, modifies it, and publishes the modified version, or when a group of people collaborate on a set of documents, synchronizing irregularly with a central server (as in CVS [12]), a network of servers (as in Lotus Notes) or directly with each other (as in Groove [20]). In each of these cases, a user should be able to work on the version at hand and then either merge it with others or fork to a different branch, as well as rollback the current changes or look at differences between versions *without network connectivity*.

The main contribution of this paper is the Storm (for *STORage Module*) design, a hypermedia system built to use the emerging peer-to-peer data lookup technologies to enhance data mobility by dealing with versioning and dangling links.

Storm is a library for storing and retrieving data as *blocks*, immutable byte sequences identified by cryptographic content hashes [29]. On top of the block layer, Storm provides services for versioned data and Xanalogical storage [34]. We address the mobility of documents by block storage and versioning, and the movement of content between documents (copy&paste) by Xanalogical storage. Fig. 1 provides an overview of Storm's components.

Additionally, we hope to provide an input to the ongoing discussion about peer-to-peer hypermedia systems [46, 7, 49, 29].

This paper is structured as follows. In the next section, we describe related work. In section 3, we give an overview of the xanalogical storage model. In section 4, we introduce the basic storage unit of our system, i.e. file-like blocks identified by cryptographic hashes. In section 5, we discuss application-specific reverse indexing of blocks by their content, essential for many applications. In section 6, we present techiques for efficient versioned storage of mutable data on top of blocks. In section 7, we report on implementation experience and future directions. Section 8 concludes the paper.

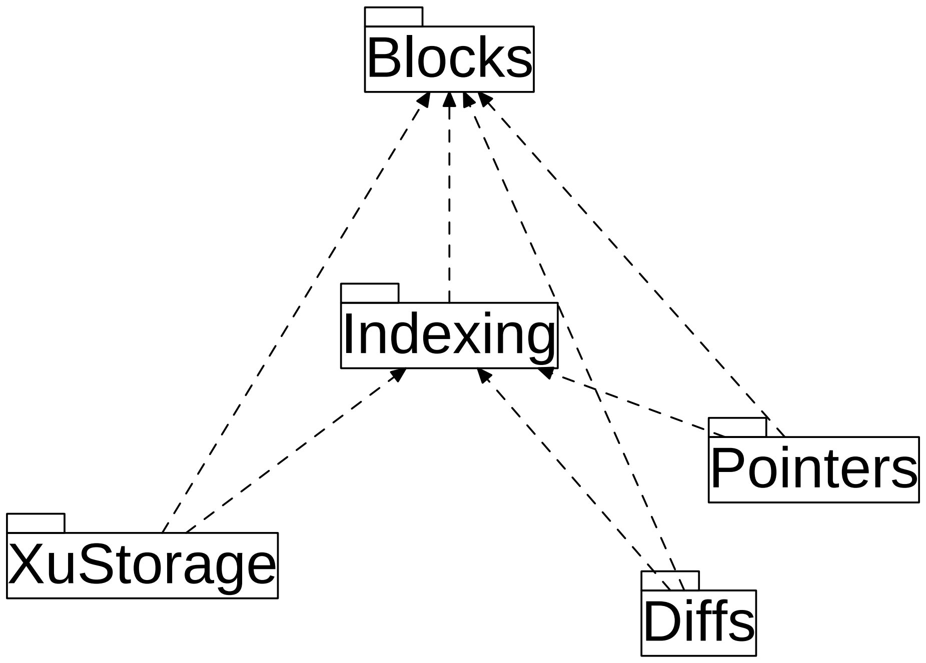

Figure 1: Components of the Storm model

2 Related Work

2.1 Dangling links and alternative versions

The dangling link problem has received a lot of attention in hypermedia research (e.g. [15]). As examples, we examine the ways in which HTTP, Microcosm [17] and Hyper-G [2] deal with the problem.

In HTTP, servers are able to notify a client that a document has been moved, and redirect it accordingly [16]. However, this is not required, and there are no facilities for updating a link automatically when its target is moved. The HTTP protocol includes a "LINK" request for creating a relationship between a set of URIs, but this feature has never been commonly implemented [39].

In Microcosm, hypermedia functionality is implemented using *filters*, which react to arbitrary messages (such as 'find links to this anchor') generated by a client application. Filters are processes on the local system or on a remote host [21]. When a document is moved or deleted, a message is sent to the filters. Linkbases implemented as filters can update their links accordingly. A client selects a set of remote filters to use. Only links stored by one of these filters can be found by the client.

In Hyper-G, documents are bound to servers, and a link between documents on different servers is stored by both servers [24]. This ensures that all links from and to a document can always be found, but requires the cooperation of both parties. Hyper-G employs a scalable protocol for notifying servers when a document has been moved or removed. A server hosting links to this document can then ask the link's author to change the link, or at least the link can be removed automatically. The *p-flood* algorithm employed by Hyper-G guarantees that a message is delivered to all interested servers, but requires that each interested server keeps a list of all the others.

These approaches share the assumption that it is not possible to resolve a location-independent identifier. Otherwise, it would not be necessary to update links when a document is moved, nor would either of the servers storing two given documents need to know the links between them; knowing only a document's location-independent identifier, it would be possible to find both the document and links to it, no matter which peer in the network they are stored on.

In a similar vein, version control systems like CVS or RCS [47] generally assume a central server hosting a repository. The WebDAV/DeltaV protocols, designed for interoperability between version control systems, inherit this assumption [19, 11]. On the other hand, Arch [27] places all repositories into a global namespace and allows independent developers to branch and merge overlapping repositories without any central control.

Lotus Notes, a popular database sharing and collaboration tool, uses both location-dependent and location-independent identifiers [28]. However, partly due to the age of the system, Lotus Notes is limited to client-server architecture. Groove [20] is an improved design based on Lotus Notes, employing strong security mechanisms and usesing peer-to-peer functionality as the basis of communication channels among a limited amount of participants.

2.2 Peer-to-peer systems

During the last few years, there has been a lot of research related to peer-to-peer resource discovery, both in the academia and in the industry [37]. There are two main approaches: broadcasting [18, 40, 26], and distributed hashtables (DHTs) [44, 38, 50, 41, 32, 31]. Broadcasting systems forward queries to all systems reachable in a given number of hops (time-to-live). DHTs store (key,value) pairs which can be found given the key; a DHT assigns each peer a subset of all possible keys, and routes queries for a given key to the peer responsible for it. Before a pair can be found, it must be *inserted* in the DHT by sending it to the peer responsible for the key. Both approaches use an application-level overlay network for routing.

While broadcasting systems' performance can be worse than linear, DHTs' performance usually has log-like bounds in the number of peers for *all* internal operations[3]. This scalability is what makes global searches feasible in DHTs. In broadcasting approaches, on the other hand, scalability is achieved by forwarding queries only to a limited subset of the

[3]It's not clear whether *all* proposed DHT designs can preserve log-like properties when participants are heterogeneous and they join and leave the system in a dynamic manner.

peers (bounded by the time-to-live), which means that searches in these systems are not truly global.

A DHT has a *key space*, for example the points on a circle. The keys in (key,value) pairs are mapped to points in the key space through a hash function. Independently, each peer is assigned a point in the space. The DHT defines a distance metric between points in the key space (e.g. numeric, XOR); the peer responsible for a hashtable key, then, is the one that is *closest* to it in the key space, according to the distance metric. A DHT peer is roughly analogous to a hashtable bucket. Queries are routed in the overlay network, each hop bringing them closer to their destination in key space, until they reach the responsible peer. A common API that can be supported by current and future DHTs is proposed in [14].

The basic definition of a distributed hashtable does not indicate how large the keys and values used may be. Intuitively, we expect keys to be small, maybe a few hundred bytes at most; however, there are different approaches to the size of values. Consider a file-sharing application: If the keys are keywords from the titles of shared files, are the values the files– or the addresses of peers from which the files may be downloaded? Iyer et al [23] call the former approach a *home-store* and the latter a *directory* scheme (they call the peer responsible for a hashtable item its 'home node,' thus 'home-store'). The choice between the schemes affects the scalability and reliability of the network.

CFS [13] and PAST [42] are scalable storage systems using the home node approach, based on the Chord [44] and Pastry [41] DHTs, respectively. Freenet [10] is a system for anonymous reading and publication.

Recently there has been some interest in peer-to-peer hypermedia. Thompson and de Roure [46] examine the discovery of documents and links available at and relating to a user's physical location. An example would be a linkbase constructed from links made available by different participants of a meeting [45]. Bouvin [7] focuses on the scalability and ease of publishing in peer-to-peer systems, examining ways in which p2p can serve as a basis for Open Hypermedia. Our own work has been in implementing Xanalogical storage [29].

At the Hypertext'02 panel on peer-to-peer hypertext [49], there was a lively discussion on whether the probabilistic access to documents offered by peers joining and leaving the network would be tolerable for hypermedia publishing. For many documents, the answer is probably no; however, for personal links, comments, and notes about documents, probabilistic access may be acceptable, especially when seen as a trade-off against having to set up a webspace account before publication.

In the end, some peers will necessarily be more equal than others: Published data will be hosted on servers which are permanently on-line, but are otherwise ordinary peers in the indexing overlay network.

3 Overview of Xanalogical storage

In the xanalogical storage model [33], pioneered by the unfinished Project Xanadu, links are not between documents, but individual characters.

When a character is first typed in, it acquires a permanent id ("the character 'D' typed by Janne Kujala on 10/8/97 8:37:18"), which it retains when copied to a different document, distinguishing it from all similar characters typed in independently[4]. A link is shown between any two documents containing the characters that the link connects. Xanalogical links are external and bidirectional.

In addition to content links, xanalogical storage keeps an index of transclusions: identical characters copied into different documents. Through this mechanism, the system can show to the user all documents that share text with the current document.

To track links and transclusions, the system indexes documents by the characters they contain, and links by the characters they refer to. To find transclusions of a document, we search the index for other documents containing any character from this document. To show links, we first search for links refering to any character in this document. However, when we have a link, we don't yet have the document it links to. Therefore, in a second step, we search for documents containing the characters the link targets.

Of course, doing any expensive operation (like an index lookup) for *every* character in a document does not scale very well. In practice, characters typed in consecutively are given consecutive ids, such as `...:4`, `...:5`, `...:6` and so on, and operations are on *spans*, i.e. ranges of consecutive characters (`...:4-6`).

Our current implementation shows only links between documents that are currently stored on the local disk (Fig. 2, discussed in [30]). In the future, we will implement a global distributed index on top of a distributed hashtable (Section 5). To find the transclusions of a span, the system will retrieve all transclusions of any span with the same prefix (`...:`), then sort out those that do not overlap the span in question.

Since the problem is to search for overlapping ranges, the spans themselves can not be used as hashtable keys. However, we keep the number of spans with the same prefix relatively small (limited by the amount of text the user enters between two saves of a document). Therefore, this should not be a major scalability problem. Otherwise, systems that allow range queries, such as skip graphs [3], may prove useful.

One question is how to select links to show when a popular document has been linked to by a large number of users. We hope to address this problem by collaborative filtering of links. Collaborative filtering in peer-to-peer systems is possible without compromising participants' privacy [8]. Prioritizing links made by others in the same working group will also be useful.

Figure 3 illustrates how xanalogical storage addresses the issue of movement of data between documents. Initially, there are documents D1 and D2, with two links (directed arrows in the figure) from D1 to two different anchors in D2, A and B. The links actually are to the *spans* A and B with permanent addresses, as illustrated by the dashed lines. Next, D2 is split in two parts, D2.1 and D2.2, moving the link targets

[4]Xanalogical storage is not limited to text. We speak about *characters* because it simplifies the explanation; picture's pixels or frames of video could be substituted.

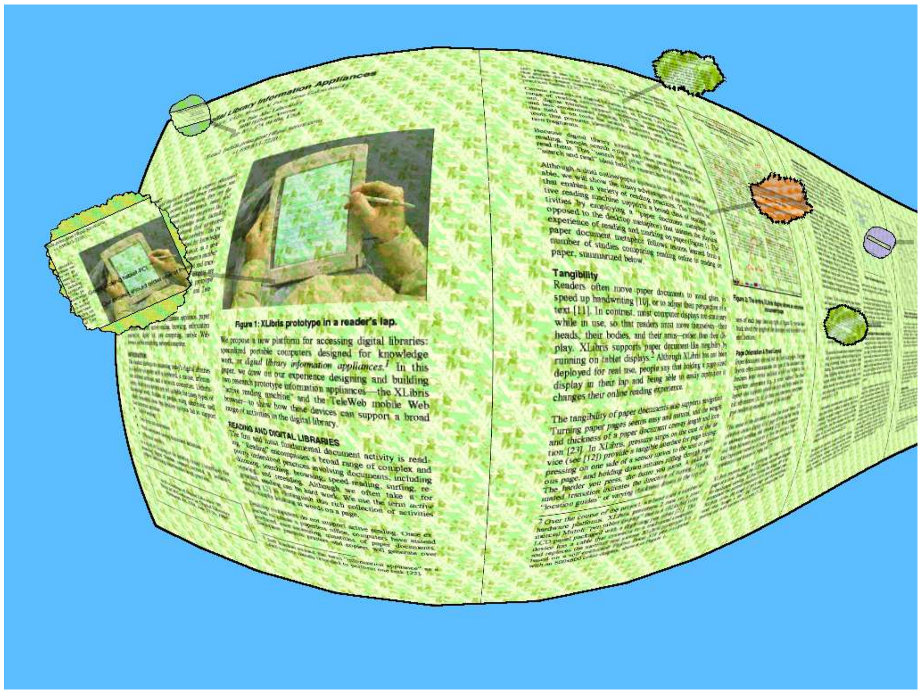
Library Information Appliances
Figure 1: XLibris prototype in a reader's lap.
READING AND DIGITAL LIBRARIES
Tangibility

Figure 2: Our current implementation showing a PDF file with links and transclusions.

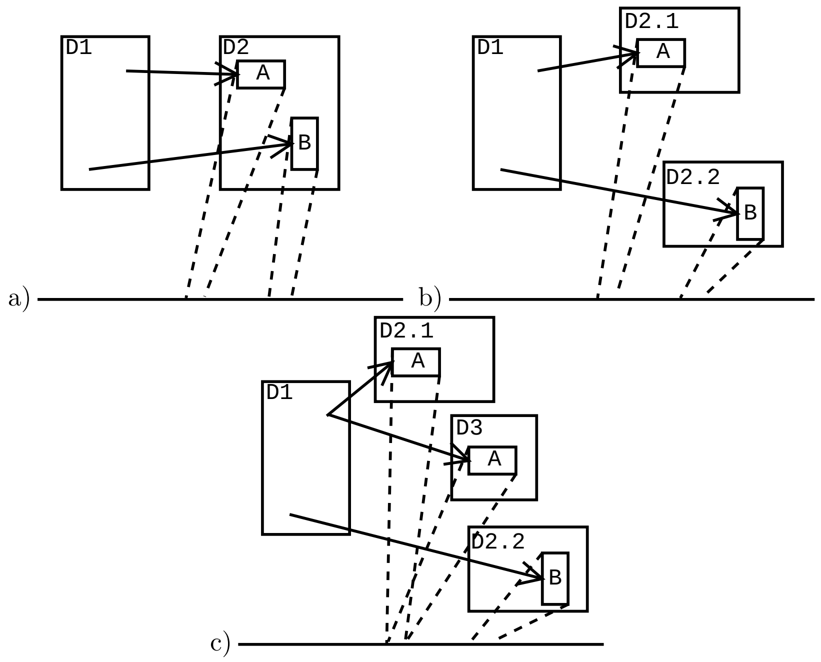

Figure 3: a) D1 links to two spans included inside D2, A and B. b) D2 is split into two. The links still point to the spans. c) A is copied into a new document. The link to A now points to both documents.

into different documents. Even though D1 is not changed, the links still work as the link targets (spans) still have the same location-independent identifiers. Finally, A is copied to a new document, D3. Again without changing D1, the link now points to both locations of A.

4 Storm block storage

In Storm, all data is stored as *blocks*, immutable byte sequences identified by a SHA-1 cryptographic content hash [35]. Purely a function of a block's content, block ids are completely independent of network location. Blocks have a similar or somewhat finer granularity as regular files, but they are immutable, since any change to the byte sequence would change the hash (and thus create a different block). Mutable data structures are built on top of the immutable blocks (see Section 6).

Storing data in immutable blocks may seem strange at first, but has a number of advantages. First of all, it makes identifiers self-certifying: no matter where we have downloaded a block from, we are able to check we have the correct data by checking the cryptographic hash in the identifier. Therefore, we can safely download blocks from an untrusted peer.

When we make a reference to a block, we can be sure that even the original author of the target will not be able to change it (unlike with e.g. digital signatures). For example, if a newspaper refers to a letter to the editor this way, the letter's sender won't be able to change the reference into an advertisement for a pornographic web page.

Secondly, caching becomes trivial, since it is never necessary to check for new versions of blocks. It is easy to replicate data between systems: A replica of a block never needs to be updated; cached copies can be kept as long as desired.

If peers make the blocks in their caches available on the network, the flash crowd problem could be alleviated: The more users request a block, the more locations there are to download it from. This resembles e.g. the Squirrel web cache [23]; however, downloads can be from *any* peer since the source does not need to be trusted. On the other hand, there are privacy concerns with exposing one's cache to the outside world.

To replicate all data from computer A on computer B, it suffices to copy all blocks from A to B that B does not already store. This can be done through a simple 'copy' command. Different versions of a single document can coexist on the same system without naming conflicts, since each version will be stored in its own block with its own id. In contrast, a system based on mutable resources has to use more advanced schemes, for example merging the changes done to a document at A or B. (Merging is still necessary when a user wants to incorporate a set of changes, but not required at replication time.)

The same namespace is used for local data and data retrieved from the network. When an online document has been permanently downloaded to the local harddisk, it can be found by a browser just as data from the network. This is convenient for offline browsing, for example in mobile environments: After a block has been downloaded, references to it will *never* cease to work, online or offline.

Thirdly, immutable blocks increase *reliability.* When saving a document, an application will only *add* blocks, never overwrite existing data. When a bug causes an application to write malformed data, only the changes from one session will be lost; the previous version of the data will still be accessible. This makes Storm well suited as a basis for implementing experimental projects (such as ours, Gzz). Even production systems occasionally corrupt existing data when an overwriting save operation goes awry; for example, one of the authors has had this problem with Microsoft Word many times.

Even when a publisher's server fails to serve a block, links to it will work until *no* other peer holds a copy. Thus, providing mirrors is trivial. Even after failure of all of the publisher's mirrors, a document may still be available from peers that have downloaded it. An archive of published blocks, in the spirit of the Web archive [22], would only be yet another backup: normal links to a block would work as long as the archive holds a copy. It would also be hard to purposefully remove a published document from the network; whether this is a good or a bad property we leave for the reader to judge.

Finally, because blocks are easy to move from system to system, we hope that block storage will be more *durable* than files. When users own multiple systems, or buy new systems to replace old ones, files are often on one harddisk and not the other, or moved to a floppy disk but not back to the harddisk. How many files you created in the 80s do you still keep around on your harddisk today? With block storage, each time a user buys a new computer, they could transfer all blocks from their existing systems to the new one, and blocks from old floppies could be copied to the harddisk without thinking about issues like which directory to keep them in. By making it easy to collect blocks produced on a diverse number of systems, it would be easier to keep old data around.

Of course, to meet this goal it is necessary that the block system remains backwards compatible at all times. We have therefore decided to enter a *persistency commitment* when we finalize the Storm design before the next release of Gzz: Any future version of the Storm specification thereafter will be able to handle any block created according to this version of the spec. This means that no matter how much we'll regret our current choices in the future, we commit to providing backward compatibility for them.

The advantages we have outlined are bought by an utter incompatibility with the dominant paradigms of file names and URLs. We hope that it would be possible to port existing applications to use Storm without too much effort, but we have not investigated the issue closely. This is because Storm was developed for the experimental Gzz system, a platform explicitly developed to overcome the limitations of traditional file-based applications.

4.1 Implementation

Storm blocks are MIME messages [6], i.e., objects with a header and body as used in Internet mail or HTTP. This allows them to carry any metadata that can be carried in a MIME header, most importantly a content type.

Collections of existing Storm blocks are called *pools*. Pools provide the following interface for injecting and obtaining data:

```
add(bytes) -> id
getIds() -> list
get(id) -> block
```

and the following methods for moving blocks between pools:

```
add(block)
delete(id)
```

Implementations may store blocks in RAM, in individual files, in a Zip archive, in a database, in a p2p network, or through other means. We have implemented the first three (using hexadecimal representations of the block ids for file names).

Many existing peer-to-peer systems could be used to find blocks on the network. For example, Freenet [10], recent Gnutella-based clients (e.g. Shareaza [43]), and Overnet [36] also use SHA-1-based identifiers. Implementations on top of a DHT could use both the directory and the home store approach as defined by [23].

Unfortunately, we have not put a p2p-based implementation into use yet and can therefore only report on our design. Currently, we are working on a prototype implementation based on UDP, the GISP distributed hashtable [25], and the directory approach (using the DHT to find a peer with a copy of the block, then using HTTP to download the block). Many practical problems have to be overcome before this implementation will be usable (for example seeding the table of known peers, and issues with UDP and network address translation [11]).

To implement *Xanalogical storage* in Storm, in each editor session we create a block with all characters entered in this session (the content type being `text/plain`). To designate a span of characters from that session, we use the block's id, the offset of the first character, and the number of characters in the span [29].

In Xanadu, characters are stored to append-only *scrolls* when they are typed. Because of this, in Storm, we call the blocks containing the actual characters *scroll blocks*. Documents do not actually contain characters; instead, they are *virtual files* containing references to spans. To show a document, the scroll blocks it references are loaded and the characters retrieved from there[5].

An important open issue with block storage are UI conventions for listing, moving and deleting blocks.

5 Application-specific reverse indexing

Finding links and transclusions in Xanalogical storage is an example of *reverse indexing* of Storm blocks: finding a block based on its contents. (For other examples, see section 6, below.) Storm provides a general API for indexing blocks in application-specific ways. We have implemented indexing on a local machine, but the interface is designed so that implementation on top of a distributed hashtable will be straight-forward. (Again, our GISP-based implementation is in a very early stage.)

In Storm, applications are not allowed to put arbitrary items into the index. Instead, applications that want to index blocks provide the following callback to a Storm pool:

```
getItems(block) ->
    set of (key, value) pairs
```

This callback analyzes a block and returns a set of hashtable items (key/value pairs) to be placed into the index. The Storm pool, in turn, provides the following interface to the application:

```
get(key) -> set of (block, value) pairs
```

[5]It is unclear whether this approach is efficient for text in the Storm framework; in the future, we may try storing the characters in the documents themselves, along with their permanent identifiers; however, this makes spoofing possible. For images or video, on the other hand, it is clearly beneficial if content appearing in different documents– or different versions of a document– is only stored once, in a block only referred to wherever the data is transcluded. This is similar to different Web pages including the same image.

This function finds all items created by this application with a given key, indicating both the application-provided value and the block for which the item was created.

We use the `getItems()` approach instead of allowing applications to put arbitrary items into the database because binding items to blocks makes it easy for pools to e.g. remove associated items when deleting a block.

As an example, the `getItems()` method of our Xanalogical storage implementation will, for a block containing a document, first collect all the spans in a document. For each span, it will then return an item mapping the span's scroll block id to the span's position in the document. When we want to find the transclusions of a span, we use `get()` to find transclusions from that span's scroll block, and load the document blocks referenced by the items.

In a networked implementation, each peer is responsible for indexing the blocks it stores. Since no peer can feasibly know all applications that may need indexing, there may be blocks available on the network that have not been indexed by a particular application. We do not see this as a problem — it's just like a block being available only if there's a peer which wants it to be — but applications must be prepared to deal with it.

Locally, on the other hand, it is guaranteed that all blocks in a pool are indexed by all applications known by the pool. To ensure this, we check that all blocks are indexed when a pool is loaded, and add missing items to the index.

One indexing application that may seem obvious is keyword-based full-text search. However, no research has been done in this direction; it is not clear whether the current interface is well suited to this, or whether current implementations are able to scale to the load to store an item for each word occuring in a document.

6 Versioning

Mutable documents can be implemented on top of block storage using a combination of two mechanisms, *pointers* and *diffs*. A *pointer* is an updatable reference to a block, and a diff is a set of differences between versions, similar to what is stored e.g. by version control systems such as CVS.

6.1 Pointers: implementing mutable resources

A Storm pointer is a globally unique identifier (usually created randomly) that can refer to different blocks over time. A block a pointer points to is called the pointer's *target* (Fig. 4).

To assign a target to a pointer, we create a special kind of block, a *pointer block*, representing an assertion like *pointer P targets block B*. To find the target of pointer P, Storm searches for blocks of this form. This

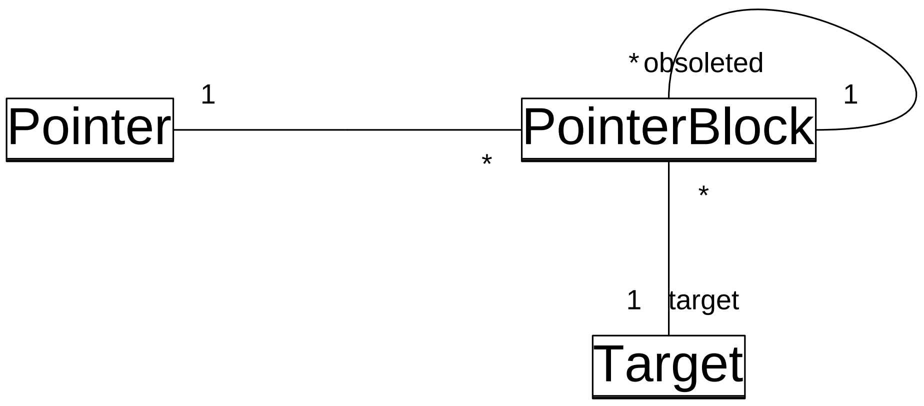

Figure 4: The Storm pointer system. A pointer is implemented by a collection of pointer blocks which can obsolete other pointer blocks and each pointer block gives a single target for the pointer.

is one application of Storm indexing (Section 5), using P as the index key.

In addition to the pointer and the target, pointer blocks contain a list of zero or more *obsoleted* pointer blocks. When a new version is created, it usually supersedes one older version; the corresponding pointer block then 'obsoletes' the pointer block targeting the superseded version. Only the new, non-obsoleted block will be considered when loading the document (although the pointer blocks pointing to past versions remain accessible for tracing the document's history)[6].

If, on the other hand, two people collaborate on a document and produce two independent versions, neither will obsolete the other. When they synchronize their pools by copying all new blocks in either to the other, both versions will be considered 'current' by the system. The users can then take appropriate action, by consolidating the changes in both versions (manually or through an automatic merge algorithm), or by treating the two versions as alternative. After the alternative versions have been consolidated, a pointer block obsoleting both consolidated previous versions is created.

Currently, the pointer mechanism works only between trusted Storm pools, e.g. in a workgroup collaborating on a set of documents. In a multi-user environment, we usually want only one user or group to be able to publish official versions a document. It is not yet clear how to do this, but digital signatures of pointer blocks seem promising. For long-term publishing, one-time signatures have been found useful [1].

The ability to retain multiple 'current' versions of a document can be useful, for example when there is no time to consolidate changes at the time of synchronization. However, we need to choose one such version when loading the document. For example, we could open an official or

[6]All known pointer blocks for a pointer are still loaded when the pointer is resolved. Storm then discards the obsoleted ones.

original version automatically if one exists.

While we think that alternative current versions are useful for asynchronous collaboration, they aren't well suited to Web-like publishing. For this, a different system may be practical, where digitally signed pointer blocks store a target and a timestamp; when resolving a pointer, the newest pointer block for that pointer would then be selected.

In summary, the current pointer system seems promising, but there are a number of unresolved issues with it: authenticating pointer blocks; the user interface for choosing between alternative current versions; and the suitability for Web-like publishing. More research is needed in this area.

6.2 Diffs: storing alternative versions efficiently

The pointer system suggests that for each version of a document, we store an independent block containing this version. This obviously doesn't scale well when we keep a lot of versions each with only small changes. Instead, we use the well-known technique of storing only the differences between versions.

We still refer to a version by the id of a block containing it. However, we do not necessarily *store* this block, even though we refer to it. Instead, we may create a *diff block*, containing the ids of two versions and the differences between them. When we want to load a version and do not have the block, we use Storm indexing to find all diff blocks from or to that version, trying to find a chain of differences starting at a known version. Then, we can apply the differences in order, and arrive at the version we seek.

When we have reconstructed this version, we create a Storm block from it and check that it matches the id of the version we are seeking. This way, we do not need to place any trust in the diff blocks we are using. While anybody can create a diff block pretending to give us version X even though it really gives us version Y, we can still retrieve diff blocks from an untrusted network source because we can check whether a block has given us version X or Y by checking the cryptographic hash.

Our current implementation is a layer above Storm block storage and indexing. This layer implements a `load(id) -> version` interface through the following simplified algorithm:

1 If the block for `version-id` is in the pool, return it.

2 Else, search for diff blocks storing the difference between `version-id` and any other version.

3 For each of these blocks, attempt to `load()` the *other* version.

4 If successful, apply the difference.

5 Check the hash of the resulting version. If correct, return it; if incorrect, go back to step 3.

As computing differences is file-format dependent, so is our system for storing versions. In our implementation, applications need to provide

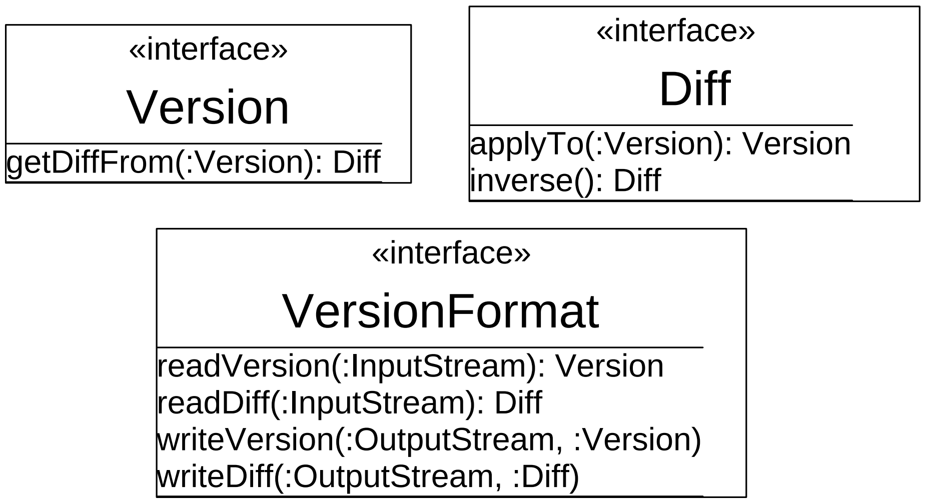
«interface»
Version
getDiffFrom(:Version): Diff
«interface»
Diff
applyTo(:Version): Version
inverse(): Diff
«interface»
VersionFormat
readVersion(:InputStream): Version
readDiff(:InputStream): Diff
writeVersion(:OutputStream, :Version)
writeDiff(:OutputStream, :Diff)

Figure 5: Diff interfaces

a callback interface for reading and writing versions and computing and applying differences.

The diff system is more complicated than simple block storage, and therefore more liable to bugs. However, saving is still purely additive: New diffs are added, but old diffs aren't changed. Therefore, when a save goes wrong, again only the changes after the previous save are lost.

To protect against buggy `Diff` or `VersionFormat` implementations, before storing a diff, we always check that we can reconstruct the appropriate version block from it; if this fails for some reason, we store the full version block instead. At the cost of some storage space, this protects the user's data.

7 Discussion

To evaluate the design, we revisit the issues raised by data mobility. For the two issues addressed, *dangling links* and *tracking alternative versions*, each individual (use) case that was identified in the introduction is dealt with here to illustrate Storm.

Dangling links. When documents are moved between servers, when using Storm the links to them are not affected as the identifiers are location-independent. In a peer-to-peer implementation, the lookup with the id returns a location where the data is currently available. If the publisher removes the document permanently, but it is archived elsewhere, the archives act as peers similarly. When there is no network connection available, but a local copy instead, Storm can find it. Also if a document and a link to it are received independently, e.g. as attachments in separate e-mails, or a link to a document in the local intranet is e-mailed, the link works. When people meet live, e.g. on a train, and form an ad-hoc network, they are able to see each other's public documents and follow links to them if a peer-to-peer implementation of Storm is used. Finally, if a document is split to parts (or content from one copy-pasted to another), links to the elements that are then in the new documents do not break, bacause with xanalogical storage the links are actually to spans that are transcluded in all the documents that show them (as illustrated in figure 3 in section 3).

Tracking alternative versions. Because Storm utilises immutable blocks, each modification to a document creates a new block. When a document is modified on several independent, unconnected systems, if there are simultaneous changes (i.e. no synching in between), there will be several versions of it. Using diffs, each version is actually (a collection of) changes to the original. What happens then, is outside the scope of Storm: the authors may decide to merge the changes forming a new joint version, but how that is done is file format and hence application specific. If (some of) the new versions of the document are not merged but forked to separate branches, they simply continue to exist (they may be assigned different names, which is again outside the scope of Storm).

8 Conclusions

We have introduced the Storm design to address two important issues raised by data mobility, dangling links and keeping track of alternative versions. In Storm, all data is stored as immutable blocks identified a SHA-1 hash. Application-specific indices of these blocks can be kept. On top of indexed blocks, we have implemented versioned storage of mutable resources as well as Xanalogical storage.

Storm is not limited to network publishing; it can be also used for private document repository. Our present implementation does not support peer-to-peer distribution yet, but the Gzz project has used it for local storage and server-based collaboration for one and a half years. Currently, we are working on a GISP-based peer-to-peer implementation.

No work on integrating Storm with current programs (in the spirit of Open Hypermedia) has been done yet. This may be difficult with Xanalogical storage; Vitali [48] notes that Xanalogical storage necessiates strong discipline in version tracking, which current systems lack. This makes Storm a rather monolithic approach at present.

One possibility is to take an existing system (with features outside the focus of Gzz) which implements strict versioning, and to modify it to use Storm for storage. A candidate is the object-oriented Web publishing environment Zope [51], which is Free Software. The open hypermedia protocol (OHP) may be another possibility [39].

Work is also needed on user interfaces for Storm.

We see Storm as a case study in the potentials of a system that does not use location-dependent identifiers at all. We hope to raise awareness for the prospects of location-independent systems based on structured overlay networks such as DHTs.

9 Acknowledgements

We would like to thank Erno Kuusela, Sarah Stehlig and Harri Oinas-Kukkonen for comments and discussions.

References

[1] R. J. Anderson, V. M. Jr., and F. A. Petitcolas. The eternal resource locator: An alternative means of establishing trust on the world wide web. In *Proceedings of the 3rd USENIX Workshop on Electronic Commerce*, pages 141–154, 1998.

[2] K. Andrews, F. Kappe, and H. Maurer. The Hyper-G Network Information System. *Journal of Universal Computer Science*, 1(4):206–220, Apr. 1995.

[3] J. Aspnes and G. Shah. Skip graphs. In *Fourteenth Annual ACM-SIAM Symposium on Discrete Algorithms*, Baltimore, MD, USA, 12–14 Jan. 2003.

[4] T. Berners-Lee. Axioms of web architecture 2: The myth of names and addresses. `http://www.w3.org/DesignIssues/NameMyth.html`, Dec. 1996.

[5] E. Bonsma. Fully decentralized, scalable look-up in a network of peers using small world networks. In *In Proc. Systemics, Cybernetics and Informatics (SCI)*, 2002.

[6] N. Borenstein and N. Freed. Mime (Multipurpose Internet Mail Extensions) part one: Mechanisms for specifying and describing the format of internet message bodies. RFC 1341, IETF Network Working Group, June 1992.

[7] N. O. Bouvin. Open hypermedia in a peer-to-peer context. In *Proceedings of the thirteenth conference on Hypertext and hypermedia*, pages 138–139. ACM Press, 2002.

[8] J. Canny. Collaborative filtering with privacy. In *IEEE Symposium on Security and Privacy*, pages 45–57, May 2002.

[9] L. A. Carr, D. C. DeRoure, W. Hall, and G. J. Hill. The distributed link service: A tool for publishers, authors and readers. In *Fourth International World Wide Web Conference Proceedings*, pages 647–656. O'Reilly & Associates, Dec. 1995. Appears in World Wide Web Journal issue 1, ISBN 1-56592-169-0, ISSN 1085-2301.

[10] I. Clarke, S. G. Miller, T. W. Hong, O. Sandberg, and B. Wiley. Protecting free expression online with Freenet. *IEEE Internet Computing*, 6(1):40–49, 2002.

[11] G. Clemm, J. Amsden, T. Ellison, C. Kaler, and J. Whitehead. Versioning extensions to webdav. RFC 3253, IETF Network Working Group, Mar. 2002.

[12] CVS. `http://www.cvshome.org/`.

[13] F. Dabek, M. F. Kaashoek, D. Karger, R. Morris, and I. Stoica. Wide-area cooperative storage with CFS. In *Proceedings of the eighteenth ACM symposium on Operating systems principles*, pages 202–215. ACM Press, 2001.

[14] F. Dabek, B. Zhao, P. Druschel, and I. Stoica. A common api for structured peer to peer overlays. Talk at OceanStore/ROC/Sahara Winter Retreat, Jan. 2003.

[15] H. C. Davis. Referential integrity of links in open hypermedia systems. In *Proceedings of the ninth ACM conference on Hypertext and Hypermedia*, pages 207–216. ACM Press, 1998.

[16] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, and T. Berners-Lee. Hypertext transfer protocol – http/1.1. RFC 2068, IETF Network Working Group, Jan. 1997.

[17] A. M. Fountain, W. Hall, I. Heath, and H. Davis. MICROCOSM: An open model for hypermedia with dynamic linking. In *Proceedings of the European Conference on Hypertext (ECHT'90)*, pages 298–311, 1990.

[18] Gnutella. http://www.gnutella.com.

[19] Y. Goland, E. Whitehead, A. Faizi, S. Carter, and D. Jensen. Http extensions for distributed authoring – webdav. RFC 2518, IETF Network Working Group, Feb. 1999.

[20] Groove networks. http://www.groove.net.

[21] G. Hill and W. Hall. Extending the microcosm model to a distributed environment. In *Proceedings of the European Conference on Hypertext (ECHT'94)*, pages 32–40, 1994.

[22] The Internet Archive: The Wayback Machine. `http://web.archive.org`.

[23] S. Iyer, A. Rowstron, and P. Druschel. Squirrel: a decentralized peer-to-peer web cache. In *Proceedings of the twenty-first annual symposium on Principles of distributed computing*, pages 213–222. ACM Press, 2002.

[24] F. Kappe. A scalable architecture for maintaining referential integrity in distributed information systems. *J.UCS: Journal of Universal Computer Science*, 1(2):84–99, 1995.

[25] D. Kato. GISP: Global information sharing protocol - a distributed index for peer-to-peer systems. In *In Proceedings of 2nd International Conference on Peer-to-Peer Computing (P2P'02)*, pages 65–73, 2002.

[26] Kazaa. http://www.kazaa.com.

[27] T. Lord. Arch. `http://www.fifthvision.net/open/bin/view/Arch/`.

[28] Lotus notes c api 5.0.3 reference guide. `http://www.lotus.com/ldd/doc/tools/c/5.0.3/api503ug.nsf`.

[29] T. J. Lukka and B. Fallenstein. Freenet-like guids for implementing xanalogical hypertext. In *Proceedings of the thirteenth conference on Hypertext and hypermedia*, pages 194–195. ACM Press, 2002.

[30] T. J. Lukka, J. V. Kujala, M. Katila, and B. Fallenstein. Buoys, break lines, and unique backgrounds for non-disruptive bidirectional spatial links. Submitted to Hypertext'03., 2003.

[31] D. Malkhi, M. Naor, and D. Ratajczak. Viceroy: a scalable and dynamic emulation of the butterfly. In *Proceedings of the twenty-first annual symposium on Principles of distributed computing*, pages 183–192. ACM Press, 2002.

[32] P. Maymounkov and D. Mazieres. Kademlia: A peer-to-peer information system based on the xor metric. In *Proceedings of IPTPS02*, 2002.

[33] T. Nelson. Xanadu® technologies– an introduction. `http://xanadu.com/tech/`, 1999.

[34] T. H. Nelson. Xanalogical structure, needed now more than ever: parallel documents, deep links to content, deep versioning, and deep re-use. *ACM Computing Surveys*, 31(4es), 1999.

[35] NIST. *FIPS PUB 180-1: Secure Hash Standard*, 1995.

[36] Overnet. `http://overnet.com/`.

[37] Peer-to-peer working group. www.p2pwg.org.

[38] S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Schenker. A scalable content-addressable network. In *Proceedings of the 2001 conference on applications, technologies, architectures, and protocols for computer communications*, pages 161–172. ACM Press, 2001.

[39] S. Reich and H. C. Davis. The need for an open hypertext protocol. *ACM SIGWeb Newsletter*, 8(1):21–23, Feb. 1999.

[40] M. Ripeanu, I. Foster, and A. Iamnitchi. Mapping the gnutella network: Properties of large-scale peer-to-peer systems and implications for system design. *IEEE Internet Computing Journal*, 6(1), 2002.

[41] A. Rowstron and P. Druschel. Pastry: Scalable, distributed object location and routing for large-scale peer-to-peer systems. In *IFIP/ACM International Conference on Distributed Systems Platforms (Middleware)*, pages 329–350, 2001.

[42] A. I. T. Rowstron and P. Druschel. Storage management and caching in PAST, a large-scale, persistent peer-to-peer storage utility. In *Symposium on Operating Systems Principles*, pages 188–201, 2001.

[43] Shareaza. `http://www.shareaza.com`.

[44] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. In *Proceedings of the 2001 conference on applications, technologies, architectures, and protocols for computer communications*, pages 149–160. ACM Press, 2001.

[45] M. Thompson, D. DeRoure, and D. Michaelides. Weaving the pervasive information fabric. In *Proceedings of the 6th International Workshop on Open Hypermedia Systems*, pages 87–95. Springer Verlag, 2000.

[46] M. K. Thompson and D. C. D. Roure. Hypermedia by coincidence. In *Proceedings of the twelfth ACM conference on Hypertext and Hypermedia*, pages 129–130. ACM Press, 2001.

[47] W. F. Tichy. RCS - a system for version control. *Software–Practice and Experience*, 15(7):637–654, 1985.

[48] F. Vitali. Versioning hypermedia. *ACM Computing Surveys*, 31(4), Dec. 1999.

[49] U. K. Wiil, N. O. Bouvin, D. Larsen, D. C. D. Roure, and M. K. Thompson. Peer-to-peer hypertext. In *Proceedings of the thirteenth conference on Hypertext and hypermedia*, pages 69–71. ACM Press, 2002.

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

[51] Zope. `http://zope.org/`.