Beyond file systems: understanding the nature of places where people store their data

Eno Thereska, Oriana Riva, Richard Banks, Sian Lindley, Richard Harper - Microsoft Research, William Odom - Carnegie Mellon University

Abstract

This paper analyzes the I/O and network behavior of a large class of home, personal and enterprise applications. Through user studies and measurements, we find that users and application developers increasingly have to deal with a *de facto* distributed system of specialized storage containers/file systems, each exposing complex data structures, and each having different naming and metadata conventions, caching and prefetching strategies and transactional properties. Two broad dichotomies emerge from this. First, there is tension between the traditional local file system and cloud storage containers. Local file systems have high performance, but they lack support for rich data structures, like graphs, that other storage containers provide. Second, distinct cloud storage containers provide different operational semantics and data structures. Transferring data between these containers is often lossy leading to added data management complexity for users and developers.

We believe our analysis directly impacts the way users understand their data, designers build and evaluate the success of future storage systems and application developers program to APIs provided by the storage systems.

1 Introduction

The My Documents folder in Windows and its equivalent on the Mac have become the default, local places within which users have stored their digital documents, photos, movies and music. This is changing, though. Increasingly users have access to this content not just on their personal devices, but also online, through file storage services such as Flickr, Facebook, Skydrive and Dropbox. They have access, also, to content not just created or owned by them explicitly, but more broadly accessible. They have instant access to *all* music ever produced, through services like iTunes, Spotify or Last.fm, and to *all* movies ever released, through services like NetFlix or Lovefilm. Online services affect not just the accessibility of content, though, but also the elemental nature of the files themselves. Photos, music and videos are no longer standalone units of content, but instead interconnected networks of streams and metadata, intrinsically connected to other pieces of content through related items, comments, review scores, tags and so on. These content relationships act to reflect the provenance and rich lifetime of the file itself.

Application developers are also affected. Increasingly they have to program not to one file system abstraction, but to a *de facto* distributed system of specialized storage containers. Each exposes complex abstractions (e.g., Btrees, key-value stores, graph stores, file stores, databases, etc.) Each has different caching and prefetching strategies (OS-based for files on desktops, browser-based and whole-object caching and prefetching for files on other containers). Naming, attributes and metadata are different in each container leading to transformation loss whenever a file moves from one container to another. Communication models (shared space vs. message passing), and atomicity and transactions across data containers differ in policy and implementation.

This paper argues that, as a research community, we do not have a set of good metrics to evaluate new storage systems built today and their impact on users and developers. Some of the metrics that served well in the past do not apply well to today's needs. Let's look at two examples. A first metric is performance. Efforts have been made in the past to replace the local file system with a database (an old example was the Inversion file system [35] and a more recent one was the failure of WinFS [45]). These have repeatedly failed, though, because performance was deemed to be inadequate. New storage systems demonstrate, however, that users are willing to be tolerant of imperfect performance as long as the system provides new experiences for them. Harter et al. show how application writers use (user-level) databases and key-value stores to augment the inadequate local file system [20]. A second metric that may no longer serve us well is the ability to support transactions. Even when these are provided (as in the case of transactional NTFS [31]) they are not pervasively used. Application developers today have to handle operations across a distributed system of storage containers and availability requirements often make transactions impractical [9]. If a handful of storage containers do not provide transactions (as is the case today) the value of transactions for those containers that provide it (like NTFS) is small.

What should the new evaluation metrics be? The main contribution of this paper is an in-depth analysis and interpretation of the I/O behavior of modern applications on emerging storage containers that we hope lays the foundation for the metrics we propose. We use user studies and measurements. For the latter, we follow the methodology taken by Harter et al. [20] and construct an *application study*. A main difference of our methodology from those used in the past is a consideration of several new storage containers that augment the local file system abstractions.

To analyze I/O and networking traces we build on the ETW tracing framework on Windows [28] that allows us to capture system calls and on HttpWatch [22] and Chrome Developer Tools [17] that allow us to capture network traffic to multiple storage containers for the Internet Explorer and Chrome browsers respectively. We examine different storage containers, which are common today. They include a traditional desktop file system (Window's NTFS), storage on Flickr, Facebook, Sharepoint, Dropbox, Last.fm, Lovefilm.com, Kindle, GoogleDocs and YouTube. We examine these containers along the axis of namespace, object attributes, access control, caching and prefetching, atomicity and transactions within and across containers, performance and sharing/communication within and across containers.

In addition to building on and confirming past findings (on file sizes and sequentially of accesses [4, 21, 36, 37, 39, 43], and complex file and database interactions on the desktop [20]), we make several new observations:

No one data structure rules: Today's storage containers expose complex data structures to manage the data. Flat files are one structure, but increasingly developers are expected to work with key-value, database and graph structures. In the past, storage systems have been evaluated on point-optimizations for one data structure (e.g., flat files). We argue that a metric for evaluating future systems would be native support for multiple, coexisting data structures.

My "stuff" has many names: To access data one needs to name it. Diverse and *ad hoc* naming strategies have emerged. Naming is not always decoupled from location. Sometimes a shared namespace illusion is provided, at other times messages are sent to containers that expose a different namespace. We argue for moving away from shared namespaces towards per-object addressing using messages across and within containers.

Transformations are lossy: Files and data are often copied or moved across storage containers. Such moves are lossy. Part of the loss comes from lack of support for multiple data structures (e.g., when copying a graph structure to a container that only supports files — the graph edges can be lost). Another part comes from lack of uniform support for attributes and metadata. A comment on a photo on Facebook might be metadata related to that photo, but it is lost when the photo is copied to NTFS. We argue for *transformation* operations among data structures to augment the copy and move operations.

Access control is unnatural: Sharing data is popular, but the access control abstraction for managing the sharing is broken. For local file systems access control dates back to single-terminal multiple-users times. For online storage containers access control can be device-transparent, but natural user expectations about sharing, taking back and possession [33] are not supported.

Caching and prefetching mismatch: Caching and prefetching are mismatched across storage containers. Most of the caching of remote objects is done by the browser, while the caching of local objects is done by the file system. The browser uses whole-object caching and can do hint-guided prefetching because it often knows about the structure of the object (e.g., it might prefetch several photos in a Facebook album). We argue for a unification of the file system and browser strategies.

Local vs. remote performance matters less: A photo stored on Flickr will take 2 orders of magnitude more latency to retrieve than a photo stored locally on the hard drive's file system. Yet, many choose to store their content on the remote services. Have we over-optimized the performance of the local file system (to be sub-10 ms)? Could the local file system enable a richer experience to developers and users by utilizing the performance slack users seem willing to tolerate?

The rest of the paper is structured as follows. Section 2 motivates this work. Section 3 provides the tracing methodology and list of applications and tasks used. Section 4 provides detailed analysis and measurements, and Section 5 discusses their implications on systems design. Section 6 reviews related work, and Section 7 concludes.

2 Background and motivation

In the past several years a trend towards storing content on online services and moving away from storage on local computers has become evident. In this section, we briefly report on a user study we conducted which served as a motivation for this work. We also illustrate our findings through a concrete example.

2.1 Summary of user study

We recruited 21 teenagers (aged from 12-17, 9 female and 12 male) from a mid-sized city in US to gain understanding into their storage habits and in particular into their relationship with online storage services. More details on the study can be found in [34].

Participants perceived online services as providing unlimited and enduring storage. They reported storing photos on their devices while waiting for the opportunity to transfer them online and often delete them locally. Files and folders created on physical devices were seen as portals to online storage. Only a small subset of local backup folders was kept. On the other hand, participants said to sometime move their 'virtual possessions' back to the material world by, for example, printing Facebook photos, often with their associated metadata.

Their attachment to content metadata was in fact evident. Participants expressed the importance of metadata in documenting their life experiences (e.g., tags or comments associated to a photo). In these cases, creating metadata added value to the data, and metadata itself became valued data. In addition to photos, participants reported using metadata for music, such as replacing album art images with personal photos or adding personal notes in playlists when giving musical playlists as a gift. Or they expressed their attachment to metadata such as the number of times a song had been played, particularly for music they do not listen to anymore, as a sort of means for revisiting past experiences. In these cases, metadata is more valuable than the actual data.

Making strong claims about how representative this user study is would require a deeper investigation and be beyond the scope of this paper. But it highlights the users' need to create, manage and share their virtual possessions, scattered across different online services. It also shows the value users place in metadata, to the point where the loss of metadata can cause more damage than that of actual data.

2.2 Example in-depth: the copy command

We now focus on a small task, a simple "copy" command that illustrates the data flow across several containers users have to deal with. This example is taken from a real usage scenario that motivated the paper. One of our colleagues had several photos stored on a university's distributed file system (AFS) and when he moved institution he "copied" the photos to various other places, including his local machine (NTFS), Sharepoint and eventually Facebook. Figure 1 shows a photo object migrating to these different containers over time. From this one user action, we describe each of the technical findings.

No one data structure rules: As the original photo

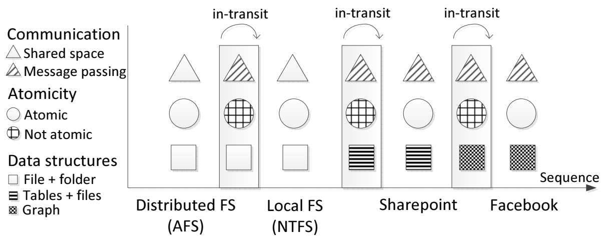

Figure 1: Moving a photo into different storage containers. Communication between files can be different within and across containers. Certain operations can be atomic within containers but are not atomic when performed across containers. Each container manages different data structures leading to lossy transformations of data across containers.

file moves from container to container, it has to be transformed to match the container's data structure. On AFS and NTFS the photo is a file, but on Sharepoint it is a record in a table, and on Facebook it is a node in a graph.

My "stuff" has many names: Naming/addressing and thus finding the photo on each of the data structures depends on the storage container. The namespace sometimes couples the name with the device/location as in NTFS; other times the namespace is device-independent as in Sharepoint or Facebook. Often, the namespace is hierarchical as in AFS or NTFS and other times it can be flat, table-based (Sharepoint) or it can follow a graph structure (Facebook). For communicating (e.g., copying) within and across storage containers, sometimes a shared namespace illusion is provided (AFS) and sometimes data has to be explicitly serialized and sent to the new location (we use the message-passing analogy found in multi-core CPU terminology).

Transformations are lossy: Different containers allow for different attributes associated with the object. In NTFS the photo itself has certain static attributes embedded inside the file. On Sharepoint, attributes associated with the photo are stored as records inside the database and are distinct objects from the photo itself. In some containers like Facebook, attributes (like comments on a photo) are as important as the object and the lines between what is data and what are attributes become blurry. When moving from NTFS to Facebook the photo file loses 94% of its attributes. When moving from Facebook to NTFS (not shown), the photo file loses 90% of its attributes (Section 4.2).

Access control is unnatural: Access control is mismatched across containers. AFS and NTFS have access control lists associated with the photo, such as who can read or write to it, but these are developed to work within a shared namespace (like the C: drive in NTFS, shared by several users, or the /afs/ namespace) and do not transform across namespaces (like Sharepoint's namespace or Facebook's). From a programmer's and user's perspective, sharing the photo with someone else if the photo is on the NTFS or AFS containers is virtually impossible. On the other hand, on Facebook, sharing a photo with someone else is a single API call. In all cases, sharing is an all-or-nothing situation: once the data is shared the sharer loses all control over it.

Caching and prefetching mismatch: Each container has a different caching and prefetching strategy and these strategies are not compatible across containers. NTFS prefetches blocks of a file for example, while Sharepoint and Facebook do whole-file prefetching. Furthermore, the latter have more hints to work with (e.g., photos in an album) and can prefetch multiple related objects. Caching is done differently too. For the applications that run on the browser, the browser determines the caching strategy, while if the application resides on the operating system, the OS determines the caching strategy.

Local vs. remote performance matters less: The latency when accessing a photo on AFS, Sharepoint and Facebook differs by three or more orders of magnitude from the local NTFS performance (Section 4). This does not stop a user from storing the data on the remote storage container because they are getting something meaningful in return (a new user experience).

No transactional expectations: Each storage container *can* support transactional properties. For example, the NTFS can be transactional [31]. Internally, once a request arrives in Facebook's servers, it can be stored atomically in databases [6]. However, these transactional properties are often just a part of the end-to-end operations, which are often not transactional (like moving a photo from NTFS to Facebook). For end-to-end transactions, a developer has to implement transactional properties when working with files across the containers (a hard task which is often not done.)

3 Methodology

We have put together a set of simple tasks representative of what users go through when storing and retrieving their data. Some tasks can be executed through traditional applications installed on a user's computer. For those, we use the ETW tracing mechanism in Windows [28] to understand the I/O behavior. This part is similar to the methodology of Harter et al. [20].

Many tasks however can only be executed through online services for which no application installation is necessary. For these tasks we are able to monitor the I/O and networking behavior by tracing the HTTP traffic from the user's device to the service through HttpWatch [22] for Internet Explorer and the Chrome Developer Tools (Network Panel) [17] for the Chrome browser. We use Internet Explorer 9.0 by default, except for the Kindle application below that requires Chrome. The service itself is treated as a black box but the APIs to store and retrieve data from it are visible to our tracing mechanism.

Application	Task and description
Word	*Create* and *edit* a document
	Permit write access to a friend
Powerpoint	*Create* and *edit* a presentation
Media player	*Open* and listen to 5 songs
Photo viewer	*Open* and view album of 20 photos
Flickr	*Upload* 20 photos from NTFS to Flickr
	Create an album with all 20 photos
	Comment on your own photo
	Permit a friend to comment on an album
	Open all photos in your friend's album
	Copy friend's album photos to NTFS
	Permit Facebook to access uploads
Facebook	*Upload* 20 photos and a movie
	Create an album with all 20 photos
	Tag the 20 photos with friends' names
	Share the album with a set of friends
	Comment on your own photo
	Copy a photo from Facebook to Flickr
Sharepoint	*Create* a folder and *upload* 10 files from NTFS
	Tag 5 documents with keywords
	Permit read or write access to a friend
	Edit a file concurrently with friend
Dropbox	*Add* an existing folder to Dropbox
	Permit write access to a friend
	Edit a file concurrently with friend
Last.fm	*Create* a playlist with 10 songs
	Open songs in playlist in shuffle mode
	Share playlist with a friend
Lovefilm.com	*Open* and watch a movie
Kindle	*Download* 5 free books
	Open the first chapter on each book
	Edit by annotating 3 paragraphs
	Share annotations with a friend
GoogleDocs	*Edit* a document by adding 5 text paragraphs
	Share a file with a friend and edit concurrently
YouTube	*Upload* 2 video clips
	Open and watch 5 video clips

Table 1: Applications and tasks used in the measurements. They include single-user, cross-container and cross-user tasks. In *italics* are common verbs/tasks.

The applications and services monitored and the user tasks are as follows. Table 1 lists the exact tasks.

Desktop suite on local file system: We use four common desktop applications storing data on the NTFS file system on Windows 7. These applications are used to validate the findings in [20] and contrast them with the cloud storage containers. Microsoft Word 2010 and Powerpoint 2010 are used to edit documents and presentations respectively. Windows Media Player is a music player capable of audio and video playback. Windows Photo Viewer is a simple photo viewer application.

Flickr: Flickr [13] is a photo store. Users store photos there because storage space is free and because it is easy to share photos among people, get feedback on photos or provide feedback to others' photos. Flickr gives developers an API to store and retrieve files, and is specialized in photo files. The tasks we perform on Flickr have to do with adding photos, creating photo collections, commenting on photos and sharing them with others.

Facebook: Facebook [11] is a service allowing users to store a range of files (photos, movies, documents) and to share them with others. It has been shown that users use Facebook as a primary store [33], thus bypassing the local file system and directly storing content there. Facebook has a developer API (the Graph API [12] and a SQL-like extension) that gives the ability to store and retrieve content, annotate content and relate it to other content. The common tasks we perform on Facebook are adding photos and movies, commenting on them and sharing them with others.

Sharepoint: Sharepoint [44] is a Wiki-like application for enterprises that allows storage and retrieval of any arbitrary media types. It has a complex API exporting database tables and file abstractions. It is mostly found in enterprise environments because it has capabilities for data life-cycle management. The common tasks we perform on Sharepoint is populating it with white-paper and presentation documents, reading and writing the documents, sharing the documents with others and creating collections of documents.

Dropbox: Dropbox [10] is an application for sharing documents and for collaborative editing. It is similar to SkyDrive [30] or iCloud [3]. The unit of storage is the traditional files and folders. Common tasks we perform on Dropbox are sharing a folder and editing files among a group of users.

Last.fm: Last.fm [24] is a service that allows streaming of any music title to any device. It is used to represent one vision of the future that imagines all the music ever created being available from anywhere (without necessarily copying them on the "local" file system). The common tasks we perform with Last.fm are creating a playlist, importing several songs there, listening to the songs and sharing the playlist with friends.

Lovefilm.com: Lovefilm.com [25] is a service that allows streaming of movies. It is used to represent one vision of the future that imagines all the movies ever created being available from anywhere (without necessarily copying them on the "local" file system). The tasks we perform with Lovefilm are watching a movie and skipping forward and backwards through the movie.

Kindle: The Kindle Cloud Reader [2] is a service that allows reading and annotation of books without necessarily copying them on the "local" file system. The tasks we perform with Kindle are reading and annotating books.

GoogleDocs: GoogleDocs [18] is a service that is similar to Microsoft's Office. Both can store and retrieve office documents, presentation and spreadsheets on the cloud. Microsoft's Office was designed for operating on a local file system and only recently allowed storing documents on the cloud, while GoogleDocs was designed with the cloud in mind from the start. The tasks we perform involve creating documents, presentations and spreadsheets, editing them and storing a copy of them locally as well as on the cloud.

YouTube: YouTube [19] is a service where anyone can upload, store and share videos. The tasks we perform with YouTube are uploading and watching videos.

4 Measurements and analysis

This section compares the storage containers along a number of axis.

4.1 Data structures and namespace

In this section, we examine the data structures and namespace that the storage containers expose. A developer creating an application that accesses data stored across these containers needs to think about the basic unit of stored data, how to name it, and how to subsequently retrieve it.

To understand the data structure and namespace affordances of each storage container, we ran the tasks in Table 1. All of them need a way to name and address objects when creating them and a way to query the store's namespace when retrieving them. The tasks are not comprehensive, and many containers offer more functionality than our tasks cover. Nonetheless, the tasks helped reveal the main results in Table 2.

In all the desktop applications running on NTFS, the basic addressable unit is a file and a folder/directory. The name of the unit is tightly coupled with the location (e.g., "C:/Users/Public/Pictures/Sample Pictures/Desert.jpg"). As reported previously [20], there are databases used on the desktop, however the native API provided by the operating system is still file-based (a database's data is ultimately stored on a file, i.e., there is no OS support for native databases).

Several containers such as Dropbox and Kindle (the app version) have a device-semi-transparent namespace. By that, we mean that their basic unit (e.g., file in Dropbox) exists and can be accessed independently of the device (from anywhere), but it is also often stored/cached on the device accessing it as a file on NTFS. The other containers are mostly location- and device-independent because they reside on the cloud. However, of course, they are not namespace-independent and the basic unit can only be found inside a namespace. There is no single global file system namespace, as proposed by [26].

The addressable unit in the containers ranges from files and folders, to opaque object IDs and data struc-

Data store	Naming and location	Addressable unit	API
NTFS (desktop apps)	Naming coupled w/ location	File, folder	File streaming
Flickr	Location-independent	Object ID, sets	Graph-like API
Facebook	Location-independent	Object ID	Graph and FQL
Sharepoint	Location-independent	Object ID, list	LINQ and proprietary
Dropbox	Device-semi-transparent	File, folder	File streaming
Last.fm	Location-independent	Song, album	Last.fm API
Lovefilm.com	Location-independent	Movie	Graph-like API
Kindle	Device-semi-transparent	Book	File streaming and SQL
GoogleDocs	Location-independent	File and collection	Google data protocol (REST-like)
YouTube	Location-independent	Video	Google data protocol (REST-like)

Table 2: Naming, addressing, data structures and APIs for different storage containers.

tures such as lists, sets and collections. For specialized containers (like Kindle, Lovefilm.com, Last.fm and YouTube), the object ID refers to their specialized data (e.g., book, movie, video) and we chose to list the data type instead of the opaque “object ID”.

The API an application developer has to use is quite diverse. On the desktop, the APIs are mostly either file streaming APIs (often wrapped in other layers such as Win32 on Windows) or (user-level) database APIs like SQL. Many of the online containers implement their set of APIs. A common feature among these APIs is a graph-like semantics (e.g., for Facebook through the Graph API and for Flickr through the Flickr API). The need to describe and query relationships among data is key for some of the online containers. This is arguably true for the desktop file system too, but the folder abstraction can only describe one kind of relationship among files. Several of the online containers also have in common a REST-like API where *PUT*, *GET* and *POST* are the basic building blocks.

Interpretation: Uniquely from other file system studies, we find that data stores export different data structures and APIs and allow for a variety of naming and addressing schemes. The set of storage abstractions has grown organically and thus it is difficult for a developer to develop applications that access data from the different containers. The desktop container has been the least innovative and exposes the same data structures and APIs now that it did 30 years ago. Harter et al. showed that “a file is not a file” and application developers want to store richer data structures on the file system [20]. Unfortunately the desktop file system cannot natively store anything else but files.

4.2 Metadata, attributes, transformations

In this section, we examine the blurring lines between data and metadata/attributes. We also look at transformations of data across different containers. For this section, the most relevant tasks in Table 1 are the ones that copy or move a file from one container to another.

The number of attributes describing an object such as a document, photo, music or video varies with the type of storage container. Figure 2 shows a representative example, with the number of attributes of a photo object when stored in the different containers. The total number of attributes for each container varies from 18 (Facebook) to 87 (Flickr)[1]. These attributes include common properties such as object size, type, date, access control settings as well as description (e.g., Exif tags), tags and comments.

The desktop file system has a clear-cut differentiation between data and metadata. On NTFS attributes can reside on a secondary file stream. Most applications however, like Media Player and Photo viewer in this particular instance often embed some attributes within the file itself. The other storage containers have a less clear-cut definition of data and metadata. For example, a photo and the associated comments on Facebook have a graph-like relationship (the photo and comments are nodes, and they are related with edges). It is not clear how to copy such a relationship to NTFS, and the transformation/copy is often lossy (more below).

Figure 2 also splits the attributes into two types. The first type, namely “attributes as values”, refers to the attributes being stored as an interpretable value. The second type, “attributes as references” refers to the attributes being stored as references to values (the analogy in programming languages is call-by-value and call-by-reference respectively). A desktop file system like NTFS stores attributes by value only. Facebook and Flickr store several attributes as references (9 and 5 respectively). Often the references are to “live” content, such as a user name attribute referring to a user’s details (Sharepoint

[1]Despite the similarity of Facebook and Flickr as photo sharing services, the main reason for such a lager difference in the number of photo attributes is that Flickr, as well as most storage containers, store Exif tags (corresponding to 60 attributes in Flickr) while Facebook does not.

From/To	NTFS	Flickr	Facebook	DropBox	Sharepoint	GoogleDocs
NTFS	0.0	47.5	94.4	1.6	4.9	3.3
Flickr	98.9	0.0	NS[2]	98.9	98.9	NS
Facebook	88.9	NS	0.0	88.9	88.9	NS
Dropbox	0.0	47.5	94.4	0.0	4.9	3.3
Sharepoint	0.0	NS	NS	0.0	0.0	NS
GoogleDocs	4.9	NS	NS	4.9	0.0	0.0

Table 3: Percentage of lost attributes when moving a photo from one storage container to another (NS means operation not supported).

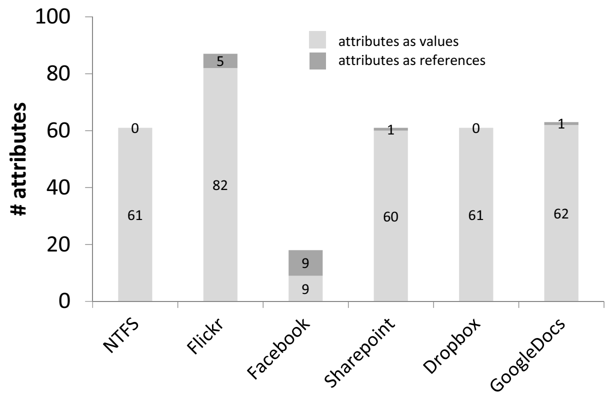

Figure 2: Number of attributes of a photo object on different photo containers. Attributes are classified as "values" and as "references" to values.

and GoogleDocs store user names as references as well). We argue that part of the value derived from a storage container comes from these non-static attributes.

As a direct consequence of the above, it is clear that certain transformations of data from container to container will be lossy. For example, when moving a photo object across containers some of the attributes are likely to be lost unless the operation occurs across containers which are file system based such as Sharepoint, Skydrive, DropBox or GoogleDocs. As Table 3 shows, when uploading a photo to Facebook, 94.4% of its attributes (e.g., Exif tags, description, dates) are lost, and still when downloading a photo from Facebook to the local file system 88.9% of the attributes (in particular tags and comments) are lost. When moving a photo from NTFS to Flickr the loss is less because most of the Exif tags (including latitude and longitude) are preserved. However, when downloading the same photo from Flickr to NTFS, Exif tags are lost and only the type attribute is preserved.

We have verified that other objects, such as videos and documents suffer similarly from transformational loss.

Interpretation: Different storage containers support different types of metadata and attributes. The difference between data and metadata is often blurred in the non-desktop containers with metadata often being of equal importance to the end-user (e.g., comments on a photo). Uniquely from other file system studies we measure the issue of data transformations across containers and find that the transformations are often lossy with respect to attributes. The next subsection shows that transformations are lossy with respect to access control as well.

4.3 Access control and sharing

In this section, we analyze the ways data can be shared with other users and applications, and the access control options available. The most relevant tasks in Table 1 for this section are the ones that share data with others or attempt to view and modify data. Table 4 provides a detailed analysis focused on photo objects only, while Figure 3 graphically summarizes the findings.

The first axis of measurement is whether one shares data with users directly or with applications and services as well. As seen in Table 4, only a few of the stores (Facebook, Dropbox and Flickr) allow access control on data shared with applications. Flickr only supports read permissions, Dropbox supports read/write permissions to only a specific application folder (sandbox access) or to all user's folders (full access), while Facebook offers 5 different types of permissions for data of the user and even for data of the user's friends.

When it comes to sharing data with users, there are several ways to name the users. NTFS uses usernames and group IDs. Flickr and Facebook have "friends", "family", "friends of friend", contacts, members of the same service, custom lists and "anyone/public". Dropbox uses user IDs or "public". Sharepoint and GoogleDocs can restrict by user ID.

The second axis of measurement is the type of permissions when sharing with users. The desktop apps based on NTFS have an extended set of permissions that controls reads, writes and deletes of both data and attributes as well as execution permissions and permissions on who can change permissions and take ownership of the data.

[2]Flickr allows automatically posting a photo on the Facebook's wall, but the photo is copied as a reference/link.

Service	Sharing with users				Sharing with apps/services		
	Unit	Permissions			Unit	Permissions	Pre-defined services
NTFS	username group	read data read attrib. read e.attrib.	write data write attrib. write e.attrib.	exec delete read perm. change perm. take ownership	NS	NS	none
Flickr	friend family contacts any member anyone	view	comment add notes add tags add people	download blog print share with others' apps	app id	read	public search Facebook Twitter Tumblr Live.Journal Wordpress Blogger
Facebook	friend friend of friend custom list public	view		share with friends' apps	app id	read (users' photos) read (friends' photos) search publish delete	public search ads instant- personalization
Dropbox	user ID public	read data read only	write data		app id	sandbox access full access	none
Sharepoint	user ID	view	edit	delete	NS	NS	none
GoogleDocs	user ID	read	write	download	NS	NS	none

Table 4: Access control options for users and applications when sharing a photo object with other users or apps/services. "e.attrib." stands for extended attributes. NS means operation is not supported.

Flickr has settings for controlling specific attributes such as comments, notes, tags, and people. In Facebook, these settings are collapsed into one setting "view". An additional setting allows the user to control access to his content by friends' applications. However, both Flickr and Facebook lack write permissions.

As observed for attributes, it is hard to move content from one container to another while fully respecting the access control options, i.e., the transformation is lossy with respect to access control too. In same cases, not the same options are supported and in others they are supported with different semantics. For instance, a photo in Flickr with permissions to add notes enabled and with permissions to add people disabled, would be difficult to translate into a NTFS container.

Interpretation: The sharing and access control abstractions, key to development of early file systems, need a complete redesign. For an application developer it is difficult to maintain the access control semantics desired by an end-user across storage containers. For users, it is difficult to reason about issues around data ownership and control when many of the storage containers they have their data into have very different and incompatible semantics. Others have shown qualitatively, through user studies, that users really care and worry about this issue [33], but due to the confusion and management overhead they rarely use the options given [5]. Perhaps subtly, the unevenness in access control leads to more (lossy) data transformations. For example, a user that needs to edit a photo stored in Flickr (no write permissions possible) must first transfer the photo to a storage container with write permissions (often the local file system), and then back again.

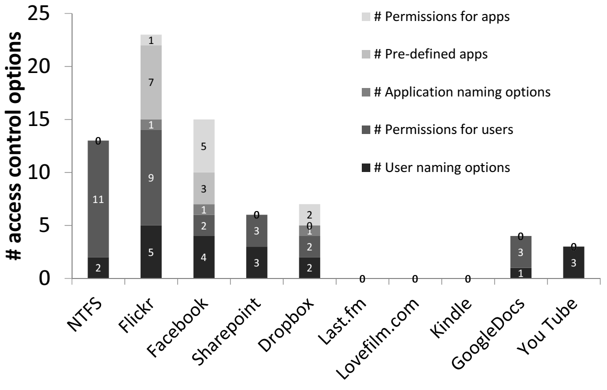

Figure 3: Access control options for different containers. Access control settings are both for sharing data with users and with external apps as well as public search engines.

4.4 Caching and prefetching

In this section, we explore the caching and prefetching strategies of the applications. Initially, we focus on the types of caching and prefetching hints that applications pass to the cache managers, the in-kernel one in case of the desktop applications and the in-browser one for web applications. Under Windows, applications may

	Hint
Word	[- ,RND, - , - , - , -]
Powerpoint	[- ,RND, - , - , - , -]
Media Player	[SEQ, - , - , - , - , -]
Photo viewer	[SEQ, - , - , - , - , -]
Flickr	[- , - , - , - ,LFT, -]
Facebook	[- , - , - , - ,LFT, -]
Sharepoint	[- , - , - , - , - , -]
Dropbox	[- , - , - , - ,LFT,CND]
Last.fm	[- , - , - , - , - , -]
Lovefilm.com	[- , - , - , - ,LFT,CND]
Kindle	[- , - , - , - , - , -]
GoogleDocs	[- , - , - , - ,LFT,CND]
YouTube	[- , - , - , - ,LFT,CND]

Table 5: Six types of caching and prefetching hints used by the applications. [SEQ,RND,NBF,WTH,LFT,CND]. SEQ stands for *FILE_FLAG_SEQUENTIAL_SCAN*, RND stands for *FILE_FLAG_RANDOM_ACCESS*, NBF stands for *FILE_FLAG_NO_BUFFERING*, WTH stands for *FILE_FLAG_WRITE_THROUGH*, LFT stands for "Lifetime" hints (such as $max - age$) and CND stands for conditional invalidation hints (such as those specified by the *ETAG* value).

pass such hints when they first create a file as flags to the *FileCreate()* call. There are four types of possible hints, allowing a developer to say that a file will be accessed sequentially, a file will be accessed in a random-access way, a file will not be accessed again so it needs no caching and that a file's changes should be immediately propagated to persistent storage (write-through).

In the browser, applications may pass hints as part of a request response together with the object. There are two types of hints, allowing a developer to specify how long an item should be cached, and whether the server supports conditional checks on an item's freshness. In a conditional check, the browser makes a request to the server to verify that a cached item has not been updated. If the item has been updated, the server sends the browser the new item, otherwise the server sends the browser a short message.

Table 5 shows the types of caching and prefetching hints and the applications employing them. In general, the desktop applications employ only the first hints, while the web applications employ only the last two hints. The web applications leave it up to the browser to decide how to store the cached items locally.

Figure 4 zooms in on the web application's two caching hints. The item lifetime hint varies by orders of magnitude between the services, even when the item type stored is the same (e.g., a photo in Facebook or Flickr). Furthermore, not all web storage containers allow for conditional caching.

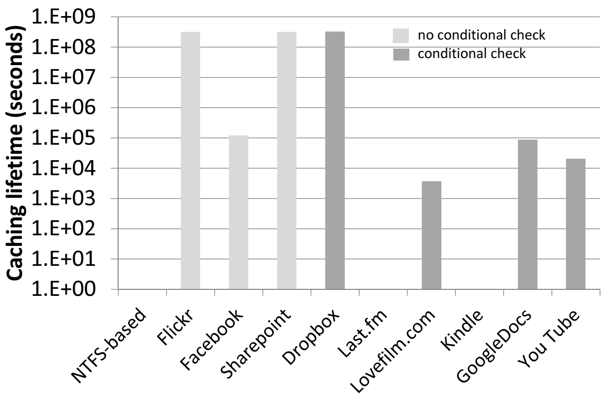

Figure 4: Caching lifetime and ability to make conditional requests. NTFS, Last.fm and Kindle do not provide any lifetime or conditional hints.

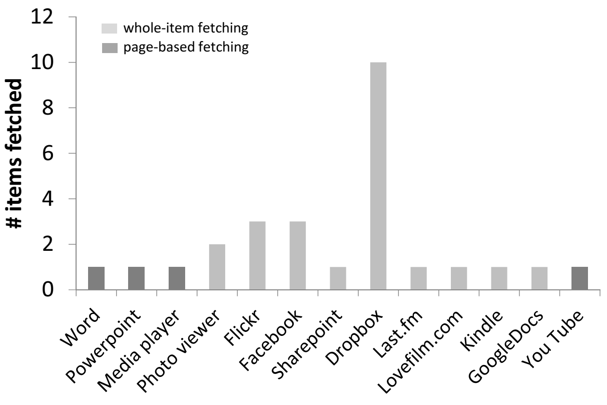

Figure 5: Different prefetching options.

Figure 5 shows the number of items fetched by the application when an item is requested. Several of the applications do deep whole-item prefetching for photos. For example, Dropbox prefetches 9 subsequent photos in a folder when a user opens the first photo. Several of the applications (in darker gray) prefetch parts of an item, e.g., as in the case of a movie on Lovefilm.com or a song on Last.fm.

Interpretation: Developers recognize that caching and prefetching hints from the application are useful to the cache manager (either in the kernel or in the browser). However, these hints are static and hardcoded, which makes them non-portable. For example, the sequential hint for an application developed for the desktop is lost if the application would run as a service on the web. Caches are not unified. Different browsers (e.g., Internet Explorer and Chrome) maintain their own caches and one browser's cache is not accessible to another's. The kernel cache manager and the browser cache manager also operate independently and without coordination.

4.5 Performance expectations

In this section, we explore the performance of the different containers. Remote storage containers often use the local file system as a building block with a distributed layer on top. These containers pay additional performance penalties from multiple network hops. As we have seen before, they also often use databases and key-value stores, which have traditionally been rejected by the local file system community [35]. How do these storage containers perform when compared with the local file system?

To get a comparable metric from containers that service different media, we focus on *first-byte latency* as a metric. This is the latency incurred from when a client makes a request to when the first byte of the response is received. This metric works well when file types are of different length (e.g., a movie vs. a document). We approximate the first-byte latency from remote service by summing up the *Blocked*, *Connect*, *Send* and *Wait* times exposed by HttpWatch or Chrome Developer Tools (Network Panel). These tools can measure precisely the time when the response is received. To approximate the first-byte latency from the local file system, on the other hand, is slightly more difficult. We could measure any *Blocked* time by subtracting the time when a disk request is initiated (*DiskIo* ETW trace) from the time when a file request was sent (*FileIo* ETW trace). The problem is that we do not have a way to measure when a *DiskIo* request is actually serviced by the disk, i.e., the disk might do its own buffering of requests and introduce other queueing delays not captured by our measurements.

As such, we go for a conservative measurement and subtract the time when a disk request is completed (*DiskIo Completed* ETW trace) from the time when a file request was sent. Request sizes are small for the applications involved (4 kB-64 kB), hence our approximation should not be that off. Figure 6 shows the results. Even with the conservatively large latency in the local case, the local file system still has the lowest average latency observed. Performance across containers varies over four orders of magnitude.

Interpretation: Developers have long used optimized local file systems for performance [27, 40, 42]. For example, the main argument against using databases as a local container was based on performance [35]. A database was perceived as slower. Non-enterprise users, however, seem able to tolerate 4 orders of magnitude loss in performance, as long as they get value out of the containers in other ways (e.g., the Facebook container allows them to express relationships among data). This interpretation is not a simple "performance does not matter" statement. It is instead a call for being mindful when evaluating a storage container solely on performance.

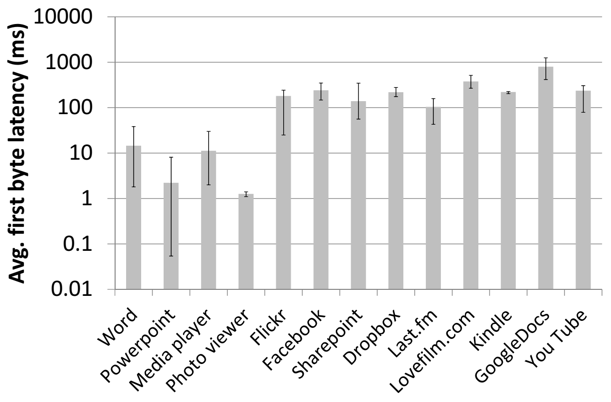

Figure 6: Performance from containers. Average, minimum and maximum latency from 5 runs is shown. Y-axis is in log scale.

5 Discussion and implications on design

Our I/O study points to several implications for future file system design.

Support for multiple data structures: We observed that different storage containers export different data structures to application developers. The local file system, for example, exports a file store, while several online storage containers export graph stores. We argue that a new metric for evaluating a new storage container could be the number of *co-existing* data structures it supports. The intuition is that the more data structures are supported natively, the easier it would be for application developers to handle data and state. A key operation that such a system has to provide would be loss-less transformation between data structures. As observed in this paper, currently such transformations are lossy (e.g., when copying a Facebook graph onto a flat file).

There are examples that show that support for multiple data structures is becoming more important. For example, the SQL Server database has recently added support for files [29], while Ren and Gibson argue for a file system to add support for key-value records [38]. SwissBox argues for cross-layer optimizations when supporting multiple data-structures [1]. For example, the caching and prefetching code could be shared but the policies could be tuned to each data structure. Thereska et al. propose using redundant replicas one for each data structure supported [41].

Distributed by default: We observed that data is stored on multiple storage containers. No one storage container holds all the data. As such, application developers and users are presented with a *de facto* distributed system of the local file systems on their devices and several cloud storage containers. However, each storage container is primarily designed as a centralized silo. We argue that a new metric for evaluating a new storage container could be around the ability to communicate with other containers. This could involve namespace management, message passing for transferring data and ability to be provenance-aware [32]. There are examples that show that inter-service communication is becoming important. Some online services today allow users to connect with other storage containers, particularly Facebook (e.g., both YouTube and Flickr allow users to have their videos and photos directly posted in their Facebook timeline) and we see a number of emerging applications [14, 23] moving into this direction.

Support for possessions: As reported by Odom et al., people are keeping a large collection of data and they are anxious about issues around ownership and possession [33]. The authors identify several design criteria to partially alleviate that anxiety. First, "knowing what you have" implies that the storage system allows the user to enumerate and interact with all their data. Second, "retaining guardianship" implies that the storage system should have a demonstrable way to prove to the users that their data is safe. Third, "giving rights or access to others" implies implementing more natural ways to do access control than the current (and, as observed in Section 4.3, mostly broken) mechanisms. Fourth, "being able to relinquish possession" implies the right to get rid of data we own. This is very hard to do today partially due to the lack of co-ordination among storage containers. As such, we highlight access control and one of the sub-areas of storage system design that requires a complete re-design.

More qualitative evaluation: We observed that traditional quantitative metrics, e.g., around performance, are insufficient to evaluate a new storage container. We argue that system designers should gently incorporate more qualitative evaluation metrics. These are necessarily "softer" metrics around new user experiences enabled (e.g., from having support for multiple data structures or the ability to relinquish possession). As systems become more reliable and reach a performance plateau, the focus should necessarily shift to these new metrics for the design of a new system.

6 Related work

We believe our study is the first to explore the storage semantics and I/O behavior of different storage containers hosted both on personal devices and in the cloud. Our methodology builds on several studies on file systems' performance and usage and studies on online services. A long series of studies on file systems starts with Satyanarayanan's analysis of files stored on the Carnegie Mellon University's file server in 1981 [39]. The metrics considered were content, size, and functional lifetime of files. Following the evolution of user applications and operating systems, a number of studies have looked into improving file systems' performance, particularly cache management [4, 21, 36, 43] and scalability [21]. These measurements have been conducted mostly in academic and research computing environments and rarely in commercial and production environments [37].

Harter et al. [20] extended the previous studies to productivity and multimedia applications on Apple desktop devices. It revealed a surprisingly complex picture of how applications handle files and persistent data, and debunked several assumptions on locality of accesses, atomic operations, role of databases etc.

We believe file systems have changed more radically than the above studies imply. The trend is for users and developers to have access not just to the music, movies, photos and documents stored on their personal device, but to most music and movies ever produced, photos ever taken and documents ever written, whether their own or belonging to others. Hence, our measurements necessarily span different storage containers and go beyond the desktop file system.

Because of this radical shift in the way users store data, some of the metrics we measure are necessarily unique from the above studies as well. We focus on the diversity of data structures and namespaces, properties of data transformations and ability to support nuanced forms of access control. Other metrics (e.g., around caching, prefetching and performance) have been discussed before but we provide new, relevant measurements.

Our study builds on studies on online social sites such as YouTube, Flickr, Facebook as well. Regarding caching and prefetching, studies have shown that YouTube videos clearly exhibit small-world characteristics thus providing additional metadata that can be exploited to improve caching at server's proxies and achieve higher download speeds [7, 8, 15]. A study on Facebook, showed strong daily and weekly temporal clusters of communication and data sharing among college students [16].

7 Summary

Users have radically changed where they store their data and how they access it. The emergence of data-structure-rich, always-online storage containers has led to a different storage architecture where the traditional desktop or single-namespace file system is just one of many participants. This paper presents detailed measurements of I/O and network behavior of a large class of home, personal and enterprise applications storing data on these diverse containers. We introduce a new set of relevant measurement metrics and discuss their implications on future storage system design.

References

[1] G. Alonso, D. Kossmann, and T. Roscoe. SwissBox: An architecture for data processing appliances. In *CIDR*, 2011.

[2] Amazon. Kindle cloud reader. `https://read.amazon.com/`.

[3] Apple. iCloud. `https://www.icloud.com/`.

[4] M. G. Baker, J. H. Hartman, M. D. Kupfer, K. W. Shirriff, and J. K. Ousterhout. Measurements of a distributed file system. *SIGOPS Oper. Syst. Rev.*, 25(5):198–212, Sept. 1991.

[5] L. Barkhuus. The mismeasurement of privacy: using contextual integrity to reconsider privacy in HCI. In *Proc. of CHI*, pages 367–376, 2012.

[6] D. Beaver, S. Kumar, H. C. Li, J. Sobel, and P. Vajgel. Finding a needle in Haystack: Facebook's photo storage. In *Proc. of OSDI*, Vancouver, BC, Canada, 2010. USENIX Association.

[7] M. Cha, H. Kwak, P. Rodriguez, Y.-Y. Ahn, and S. B. Moon. I tube, you tube, everybody tubes: analyzing the world's largest user generated content video system. In *Proc. of IMC*, pages 1–14. ACM, 2007.

[8] X. Cheng, C. Dale, and J. Liu. Statistics and social network of YouTube videos. In *16th International Workshop on Quality of Service*, IWQoS '08, pages 229–238, June 2008.

[9] B. F. Cooper, R. Ramakrishnan, U. Srivastava, A. Silberstein, P. Bohannon, H. Jacobsen, N. Puz, D. Weaver, and R. Yerneni. PNUTS: Yahoo!s hosted data serving platform. In *Proc. of 34th VLDB*, volume 1, pages 1277–1288, 2008.

[10] Dropbox. Dropbox. `https://www.dropbox.com`.

[11] Facebook. Facebook. `http://www.facebook.com/`.

[12] Facebook. Graph API. `http://developers.facebook.com/docs/reference/api/`.

[13] Flickr. Flickr. `http://www.flickr.com/`.

[14] Friday. `http://www.fridayed.com`.

[15] P. Gill, M. Arlitt, Z. Li, and A. Mahanti. YouTube traffic characterization: a view from the edge. In *Proceedings of the 7th ACM SIGCOMM Conference on Internet Measurement*, IMC '07, pages 15–28. ACM, 2007.

[16] S. A. Golder, D. M. Wilkinson, and B. A. Huberman. Rhythms of social interaction: Messaging within a massive online network. In *3rd International Conference on Communities and Technologies*, volume 30, pages 1–16, 2007.

[17] Google. Chrome developer tools: Network panel. `https://developers.google.com/chrome-developer-tools/docs/network`.

[18] Google. Google docs. `https://docs.google.com/`.

[19] Google. YouTube. `http://www.youtube.com/`.

[20] T. Harter, C. Dragga, M. Vaughn, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. A file is not a file: understanding the I/O behavior of Apple desktop applications. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, SOSP '11, pages 71–83, New York, NY, USA, 2011. ACM.

[21] J. H. Howard, M. L. Kazar, S. G. Menees, D. A. Nichols, M. Satyanarayanan, R. N. Sidebotham, and M. J. West. Scale and performance in a distributed file system. *ACM Trans. Comput. Syst.*, 6:51–81, February 1988.

[22] HttpWatch. Httpwatch. `http://www.httpwatch.com/`.

[23] Jolicloud Me - Your Personal Cloud. `http://www.jolicloud.com/me`.

[24] Last.fm. Last.fm. `http://www.last.fm/`.

[25] Lovefilm.com. Lovefilm.com. `http://www.lovefilm.com/`.

[26] D. Mazières, M. Kaminsky, M. F. Kaashoek, and E. Witchel. Separating key management from file system security. In *Proc. of SOSP*, pages 124–139, Charleston, South Carolina, United States, 1999. ACM.

[27] M. K. McKusick, W. N. Joy, S. J. Leffler, and R. S. Fabry. A fast file system for UNIX. *ACM Trans. Comput. Syst.*, 2(3):181–197, Aug. 1984.

[28] Microsoft. Event tracing. http://msdn.microsoft.com/.

[29] Microsoft. Filestream overview. `http://technet.microsoft.com/en-us/library/bb933993.aspx`.

[30] Microsoft. Skydrive. `https://skydrive.live.com/`.

[31] Microsoft. Transactional NTFS (TxF). `http://msdn.microsoft.com`.

[32] K.-K. Muniswamy-Reddy, P. Macko, and M. Seltzer. Provenance for the cloud. In *Proc. of FAST*, San Jose, California, 2010. USENIX Association.

[33] W. Odom, A. Sellen, R. Harper, and E. Thereska. Lost in translation: Understanding the possession of digital things in the cloud. In *CHI '12: Proceedings of the International Conference on Human factors in Computing Systems*, Austin, TX, 2012.

[34] W. Odom, J. Zimmerman, and J. Forlizzi. Teenagers and their virtual possessions: design opportunities and issues. In *Proc. of CHI*, pages 1491–1500, 2011.

[35] M. A. Olson. The design and implementation of the inversion file system. In *Proc. USENIX Winter Conference*, 1993.

[36] J. K. Ousterhout, H. Da Costa, D. Harrison, J. A. Kunze, M. Kupfer, and J. G. Thompson. A trace-driven analysis of the UNIX 4.2 BSD file system. In *Proc. of SOSP*, pages 15–24, New York, NY, USA, 1985. ACM.

[37] K. K. Ramakrishnan, P. Biswas, and R. Karedla. Analysis of file I/O traces in commercial computing environments. In *Proceedings of the 1992 ACM SIGMETRICS joint international conference on Measurement and modeling of computer systems*, SIGMETRICS '92/PERFORMANCE '92, pages 78–90, New York, NY, USA, 1992. ACM.

[38] K. Ren and G. Gibson. TableFS: Embedding a NoSQL database inside the local file system. Technical Report CMU-PDL-12-103, Carnegie Mellon University, May 2012.

[39] M. Satyanarayanan. A study of file sizes and functional lifetimes. In *Proceedings of the eighth ACM symposium on Operating systems principles*, SOSP '81, pages 96–108, New York, NY, USA, 1981. ACM.

[40] A. Sweeney, D. Doucette, W. Hu, C. Anderson, M. Nishimoto, and G. Peck. Scalability in the XFS file system. In *Proc. of the 1996 Annual Conference on USENIX Annual Technical Conference*, ATEC '96, San Diego, CA, 1996. USENIX Association.

[41] E. Thereska, P. Gosset, and R. Harper. Multi-structured redundancy. In *Proc. of HotStorage*, Boston, MA, 2012.

[42] E. Thereska, J. Schindler, J. Bucy, B. Salmon, C. R. Lumb, and G. R. Ganger. A framework for building unobtrusive disk maintenance applications. In *Proc. of FAST*, San Francisco, CA, 2004. USENIX Association.

[43] W. Vogels. File system usage in Windows NT 4.0. *SIGOPS Oper. Syst. Rev.*, 33(5):93–109, Dec. 1999.

[44] Wikipedia. Microsoft Sharepoint. `http://en.wikipedia.org/wiki/Microsoft_SharePoint`.

[45] Wikipedia. Winfs: Windows future storage. `http://en.wikipedia.org/wiki/WinFS`.