University of Crete
Computer Science Department

Parasitic Storage Systems: Free, Reliable and Globally Accessible Gigabytes

Nikolaos Nikiforakis

Master's Thesis

June 2009
Heraklion, Greece

University of Crete
Computer Science Department

Parasitic Storage Systems: Free, Reliable and Globally Accessible Gigabytes

Thesis submitted by

Nikolaos Nikiforakis

in partial fullfilment of the requirements for the
Master of Science degree in Computer Science

THESIS APPROVAL

Author: ______________________________
Nikolaos Nikiforakis

Committee approvals: ______________________________
Evangelos P. Markatos
Professor, Thesis Supervisor

____
Sotiris Ioannidis
Associate Researcher at FORTH

____
Apostolos Traganitis
Professor

Departmental approval: ______________________________
Panos Trahanias
Professor, Chairman of Graduate Studies

Heraklio, June 2009

Abstract

The commercialization of the Internet, the fast growth of its population and our society's increasing reliance on it as a means of information exchange, triggered the design and implementation of big, fast and reliable web services such as mail services, picture galleries and file hosting servers. Most of these services are independent of each other, usable only through their custom web interfaces and viewed as islands of functionality from their various users.

In this thesis we investigate an alternate view of these services. We utilize them as connected components of a new storage service instead of isolated entities. We design a system that acts as a storage *parasite* of the various free online infrastructures and aggregates their functionality into a new, free storage service. We implement an abstract API based on generic `put` and `get` operations which transcend each service's specific mechanisms and web interfaces. We use our framework to create a user-level file system, *ParasiticFS*, which can mount several distinct web services under a single unified namespace. All files shown locally in the ParasiticFS are in reality stored in various online services taking advantage of each service's unique functionality. ParasiticFS provides zero-cost storage that is globally accessible and characterized by high reliability and availability. Moreover, our system promotes a set of unique advantages such as NFS-like semantics, disaster recovery mechanisms and instant recovery of deleted files.

The resulting system and its performance suggests that our storage parasite solution can be practically used as a free and efficient backup service, a media filesystem or an all-purpose network storage solution.

Supervisor: Professor Evangelos Markatos

GR

Περίληψη

Η εμπορευματοποίηση του διαδικτύου, η ραγδαία αύξηση των χρηστών του και η συνεχώς αυξανόμενη εξάρτηση της κοινωνίας μας για επικοινωνία μέσω διαδικτύου αποτέλεσαν την κινητήριο δύναμη πίσω από τον σχεδιασμό και την υλοποίηση μεγάλων, γρήγορων και αξιόπιστων υπηρεσιών διαδικτύου όπως υπηρεσίες ηλεκτρονικού ταχυδρομείου, συλλογές φωτογραφιών και υπηρεσίες αποθήκευσης αρχείων. Οι περισσότερες από αυτές τις διαδικτυακές υπηρεσίες είναι ανεξάρτητες η μία από την άλλη, χρησιμοποιούνται μόνο μέσω των δικών τους παραμετροποιημένων διεπαφών και αντιμετωπίζονται από τους διάφορους χρήστες ως διακριτές και ασύνδετες μονάδες λειτουργικότητας.

Σε αυτήν την εργασία διερευνούμε μια εναλλακτική οπτική γωνία για τις υπηρεσίες διαδικτύου. Αντί να τις αντιμετωπίζουμε ως διακριτές οντότητες, τις χρησιμοποιούμε ως συνδεδεμένα μέρη μιας νέας υπηρεσίας αποθήκευσης δεδομένων. Σχεδιάζουμε ένα σύστημα το οποίο συμπεριφέρεται ως ένα παράσιτο αποθήκευσης δεδομένων εις βάρος των διαφόρων δωρεάν υπηρεσιών διαδικτύου. Το σύστημα αυτό συναθροίζει την λειτουργικότητα αυτών των υπηρεσιών σε μία νέα δωρεάν υπηρεσία ανάκτησης και αποθήκευσης δεδομένων. Υλοποιούμε ένα αφαιρετικό API βασιζόμενο σε γενικευμένες `put` και `get` λειτουργίες οι οποίες υπερβαίνουν τις παραμετροποιημένες διεπαφές κάθε υπηρεσίας διαδικτύου. Χρησιμοποιούμε το προγραμματιστικό μας πλαίσιο για να δημιουργήσουμε ένα σύστημα αρχείων επιπέδου-χρήστη, ParasiticFS, το οποίο προσαρμόζει ξεχωριστές υπηρεσίες δικτύου κάτω από έναν, ενοποιημένο ονοματοχώρο. Όλα τα αρχεία που παρουσιάζονται ως τοπικά μέσω του ParasiticFS είναι στην πραγματικότητα αποθηκευμένα σε διάφορες υπηρεσίες διαδικτύου χρησιμοποιώντας πλεονεκτικά τα χαρακτηριστικά της κάθε υπηρεσίας. Το ParasiticFS προσφέρει δωρεάν αποθηκευτικό χώρο, ο οποίος είναι προσβάσιμος από όλο το διαδίκτυο και χαρακτηρίζεται από υψηλή αξιοπιστία και διαθεσιμότητα. Επίσης, το σύστημα μας προάγει ένα σύνολο μοναδικών προτερημάτων όπως σημασιολογικά στοιχεία παρόμοια με αυτά του NFS, μηχανισμούς επαναφοράς μετά από καταστροφή και στιγμιαία ανάκτηση διεγραμμένων αρχείων .

Το προκύπτον σύστημα και οι μετρήσεις του, προτείνουν την πρακτική χρήση του παρασιτικού συστήματος μας ως ένα δωρεάν και αξιόπιστο σύστημα αποθήκευσης εφεδρικών αντιγράφων, ένα πολυμεσικό σύστημα αρχείων ή μια γενική λύση για την αποθήκευση δεδομένων στο διαδίκτυο.

Επόπτης: Ευάγγελος Μαρκάτος, Καθηγητής

Acknowledgments

I would like to thank my Supervisor, Professor Evangelos P. Markatos for his valuable insights and continuous support in my work and studies. I feel grateful to Elias Athanasopoulos who gave me the opportunity to work on this subject and whose contribution was a key for writing this thesis. I also feel grateful to Dr. Kostas Magoutis for sharing his expertise on the subject and for providing me with great ideas.

I would like to thank Dr. Sotiris Ioannidis for his support and constructive criticism over the last two years and all the current and past members of the Distributed Computing Systems Group, Vasilis Pappas, Giorgos Vasiliadis, Spyros Ligouras, Alexandros Kapravelos, Antonis Papadogiannakis, Andreas Makridakis, Demetres Antoniades, Iason Polakis, Michalis Polychronakis, Spyros Antonatos, Eleni Gessiou, Vasilis Lekakis, Giorgos Kondaxis, Giannis Velegrakis, Manos Athanatos, Michalis Foukarakis, Christos Papachristos and Manolis Stamatogiannakis for their support and friendship.

Finally, I would like to thank my parents, Theodoros and Aikaterini, and my aunt Dorothy for their support, patience and encouragement during all these years.

Contents

List of Figures

List of Tables

1 Introduction

The growth in the amount of personal digital content (photos, movies, music) and our society's increasing reliance on the Internet as a means of information exchange have fueled a plethora of Web services, offering users free access to e-mail, pictures galleries, general files, and other persistent content. To satisfy the need for storing ever more content, these services compete to provide users with increasing amounts of storage. Often their business model (e.g., income from advertising) is such that it allows them to offer this storage at no charge.

In 2004 Google launched Gmail [31], a mailing service with an initial storage capacity of 1 GB, much more than the 25 or 50 Megabytes available in most mailing services at the time. Today Gmail offers more than 7 Gigabytes for email storage per user, much more than most of its users actually use [27,28]. In 2005, Yahoo acquired the company that created flickr. Today, flickr hosts more than 3 billion user images [30]. A last category of services worth mentioning are file-hosting services. In 2006, Rapidshare was founded in Germany and became a popular choice for people wanting to share large files with other Internet users. In April 2008, Rapidshare reported that it had a storage capacity of 5.4 Petabytes for their users. [32]

A user of the aforementioned services and all their variants has access to a pool of tens to potentially hundreds of gigabytes of storage. However, today users are utilizing these services independently of one another, as islands of information and storage.

If a user wants to upload a set of pictures, she visits flickr and uses their webpage or their API to upload and tag her pictures. If she wants to upload and send large files to her friends she visits Rapidshare (or a similar file-hosting service) and uses their interface for her needs. Finally, if a user wants a large mailbox capable of holding all her emails and attachments she registers an account on Gmail. While these services have common parts (most of them use the HTTP protocol, they all have backends which can provide ample storage) they are treated as if they don't.

In this thesis we argue that an abstraction can be deduced that unifies the use and management of storage underlying all of these web services. Looking beyond each service's specifics, we observe that each service provides an upload/download mechanism through which users can store their data. This observation leads us to design and implement a system that acts as a storage parasite, taking advantage of the collective storage of all web services. We term our file system *parasitic* because it takes favor of its hosts, often in ways unintended by them, for access to their underlying storage resources at zero cost.

Through this system, users no-longer worry where to upload their data and if their format is compatible with a particular service. To achieve this we design an abstract API based on generic `put` and `get` operations that hide the complexity of communicating with each service from the programmer. Based on this API we implement our user-level filesystem, ParasiticFS.

A user can mount ParasiticFS just like any standard filesystem and immediately get access to files stored in all of the underlying web services - Fig 1.1. Like standard filesystems, ParasiticFS supports `add`, `rename`, `delete`, `move` files, creation of directories and management of remote files, as if they were stored on a local disk. In addition, due to the familiar filesystem semantics, applications that work with local files need no modification to work with ParasiticFS. For example, a document editor can open and edit a file remotely stored on Gmail or a video player can start playing a movie that is hosted on Rapidshare.

ParasiticFS's exploitation of persistent-content web services to provide a large-capacity storage abstraction at no cost does not come without challenges. For one thing, web-based native storage services offering a block storage interface to infrastructure applications such as file systems, exist (e.g., Amazon S3) and are popular but are provided for a fee (both on GB-month and Internet bandwidth-used basis). Also, persistent-content web services often place limitations on how they are used, specifically to deter unintended use. The approach of ParasiticFS is to not over-rely on any single web service but rather to leverage the existence of several services, which is a trend currently supported by market growth. In this manner, ParasiticFS aims to attain service levels comparable with commercial Web-based storage services, without the associated cost.

To summarize, the contributions of this thesis, are the following:

- We develop an abstract API that hides the complexity of communicating with each particular service, by providing the programmer with `put` and `get` operations.

- Using this API, we design and implement a fully functional user-level prototype of ParasiticFS.

- Using the ParasiticFS prototype to access aggregated storage, we show that major web services do not place limits on storing large amounts of data in short periods of time, matching our desired requirements.

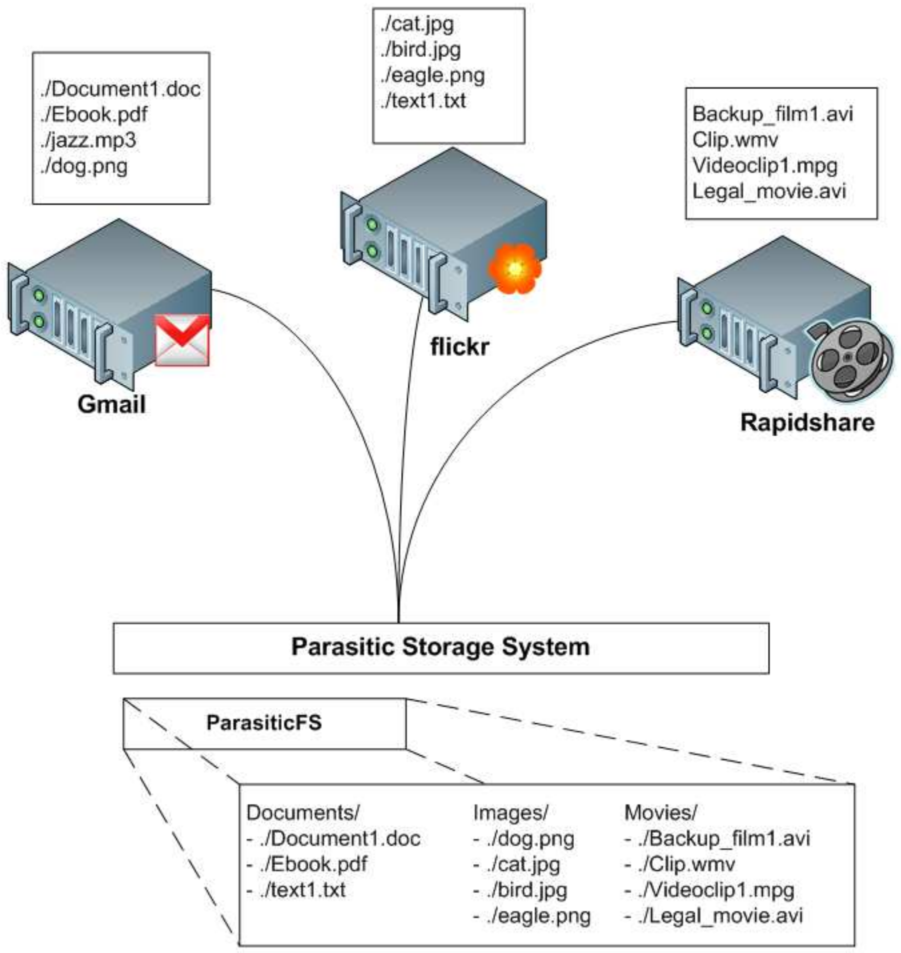

FIGURE 1.1: High level architectural view of the Parasitic Storage System

1.1 Thesis Organization

In the remainder of this thesis, we first present the various web services available to a user today and how these services can be successfully used by our storage parasite system. In Section 3 we present the reasons that motivated us to do research on a parasitic storage system. Section 4 includes and a design overview of our system followed by implementation details of a working prototype in Section 5. We evaluate our resulting system (Section 6), list the inherent and temporary limitations of our storage parasite in Section 7 and in Section 8 we explore how our current work differs from prior work in network storage and parasitic behavior systems. Finally, this thesis concludes with a summary of its main characteristics.

2

Background

In this chapter we briefly describe the various web services that can be potentially used by our storage parasite. We discuss their advantages, their disadvantages, the policies that they use in order to protect their infrastructures from misuse and how these policies can be overridden. Finally we list some commercial cloud storage solutions available today.

2.1 Candidate Web Services

Any service that can keep user data in any format can potentially be exploited by our storage parasite so that it stores data for our system.

The first, most obvious, candidate are file-hosting web services. At the time of this writing there are more than 100 different web services providing file hosting [7]. The business models behind these web services generally provide two kinds of service: a paid service and a free one. The free service usually restricts the available capacity for each “free” user and limits the upload / download bandwidth. In addition that, some paid services also restrict API access to the free users. APIs of file-hosting services enable the user to take advantage of their storage services without the hassle of visiting their website and using their custom HTML interface. These services are good candidates because they are designed to store files, and that is exactly what our parasitic storage system is trying to do. There is no need to transcode the data from one format to another or break the file abstraction by storing data in other formats. The aggregated available storage provided by a handful of these services can easily provide a free user with tens of Gigabytes for storing her files. The hassle is that the user will have to keep a list of her uploaded files and use each website’s custom interface when she needs to upload new data or download existing files.

Another type of service that is utilized by users today, is electronic mail. Lots of companies create their own email services and provide them to Internet users for free. Instead of providing free and paid versions of their services, as file-hosting services do, they make profit by placing ads on the webpages that a subscriber of their service uses. Some services, such as Gmail, use the information of each email received or sent by the user in order to show more personalized and targeted advertisments. A storage parasite can use these services to store files as attachments to emails or even as mail text.

The third and last type of web services that we describe in this section are online picture galleries. Online picture galleries are free services where users can subscribe and upload pictures. These pictures can then be shared with the user’s friends or they can be declared as public and shared with the rest of the world. Most companies providing this type of service provide it both as free and as paid with the latter having more benefits (upload space, fully functional API) than the former. In addition, their websites and APIs are focused on serving specific types of files (image files) which enables them to provide a richer and more functional API. For example, flickr provides functions which return the thumbnails of the requested images, their resolution, tags etc.

Two important properties that mailing services and picture galleries have but file-hosting services have not, are: static data location and implicit file permissions.

2.1.1 Static Data Location

When a user uploads a file to an existing file-hosting service, the service generates a unique identifier which is the user's pointer to his file. According to the various web services FAQ section, this prevents other users from guessing URLs and downloading files that do not belong to them. This is a privacy-by-obscurity technique that works fairly well when the generated identifier is of sufficient length that it cannot be easily brute-forced and when the user keeps the URL private. These unique URLs do not follow any publicly-known algorithm and thus the user cannot know in advance the id that will be appointed to his file. We call this random-id mechanism, Random Data Location (RDL). This RDL can be a good system property for a user that values his privacy but at the same time it adds to the complexity of keeping track where a user's files are. The user cannot reconstruct the pointer to his remote file if he loses the id appointed to him.

On the other hand, mailing services provide Static Data Location (SDL) since a user can always search through her inbox to find the email containing the attachment of interest to her. As described later in the ParasiticFS at least one file, the file containing the mappings between local and remote files must always be statically locateable by the storage parasite. Thus, we choose to save it on a service providing Static Data Location so that various installations of the storage parasite can always look at the same place and always find the file containing the mappings.

2.1.2 Implicit File Permissions

All modern file systems have methods of administering permissions and access rights to specific users and groups of users. This enables a file system to be multi-user and protect each user's privacy at the same time, by not allowing users (other than the intendent ones) to access a specific user's files. We can simulate such permissions on our parasitic storage system, by adding files to web services which need to authorize a user before giving access to her files instead of file-hosting services, which give away the file to anyone knowing or guessing the correct pointer. Mail services are a prime example of this service since they allow a user to view her INBOX only after she proves that she owns that mailbox (through the use of the correct credentials). This way, a user who is concerned about the privacy of her documents can place her files as attachments on her mailbox instead of uploading them to a file-hosting service.

2.2 Web Service Usage Policies

Nearly all web services impose limitations on their use so that users will not be able to exploit or overuse their service. In this section we describe the policies of three different file-hosting services, Rapidshare, DepositFiles and Sendspace [6,22,32]. File-hosting services go to great lengths to enforce policies on their usage because they are the easiest kind of service to misuse. Most file-hosting services provide accounts to members that pay a subscription fee and no accounts to free users. The websites can be used without user accounts which means that if they didn't limit their account-less use, a flash crowd [3] consisting of a few thousands free users could bring their services to a halt. Their policies generally fall under two categories: inactive file deletion and time-wait penalty between consecutive downloads.

2.2.1 Policy study # 1

Rapidshare is probably the most common service utilized by Internet users when they want to exchange large files. Their policy and limitations evolve over time but the basis of their limiting mechanisms remains the same. In Rapidshare, all users (paying and non-paying) are equal when it comes to uploading. Everyone can upload as many files as she likes without any restrictions, other than the bandwidth provided by Rapidshare for upload at the time of the transfer. However, once a file is stored on their servers and a user wants to download it, a different process is initiated based on whether the user trying to download a particular file is a subscribed user or a free user. If the user is subscribed and has an active session cookie, the download immediately starts as would happen with downloads from all servers on the Internet. If the user is not subscribed then instead of a direct download link, she is presented a page where she has to state that she is a "free user" and then she has to wait a variable number of seconds for the download link to appear. Once the file is downloaded, the free user cannot download anything else from the service until an, also variable, time-wait penalty expires. What's interesting, is the fact that the time penalty doesn't change based on the filesize of the previous download. A text file consisting of a few Kbytes and a media file consisting of hundreds of megabytes impose similar time-wait penalties.

The last policy of Rapidshare is that files that have not been accessed for a predefined number of days are deleted to relieve the system of forgotten and not needed content. This means that any reliable storage system working over Rapidshare has to keep timers for each inactive file and download it before the inactivity period expires so that the file isn't deleted by the system.

2.2.2 Policy study # 2

Depositfiles is a clone of the previously described Rapidshare service. While the layout of their website is very similar to Rapidshare's and they use the same mechanism for uploading files they differentiate themselves in file downloads. More specifically, they have country-specific download slots where all free users are placed. According to their FAQ section they use geographical IP address databases for finding out the country from which a user requests a download. When all slots for a specific country are filled, the user is instructed to try again later. They also use an inactive file-removal policy but their predefined inactivity period is shorter than Rapidshare's. The existance of download-slots mechanism, their "fast" removal policy and their low upload / download rates (see Evaluation section) suggest that Depositfiles has an inferior storage infrastructure compared to the Rapidshare infrastructure.

2.2.3 Policy study # 3

Lastly we take a look at the Sendspace file hosting service. Sendspace provides a better upload/download rate than both Rapidshare and Depositfiles and it is at the same time the least restrictive when it comes to free users. Free users can upload and download files without any time penalty between consecutive downloads. They use Javascript to ensure that a requested download originated from a browser, however their logic is flawed enabling automated downloads. They too have a removal policy for inactive files and they provide an upload API for free users. In the later sections of this thesis, we tend to use Sendspace as our web service of choice because of its high performance and lack of restrictions.

2.3 Overriding Restrictive Policies

In this section we describe some methods which can be used to override restrictive usage policies set by web services.

2.3.1 Steganography

Steganography is the method of writing hidden messages in such a way that no-one, apart from the sender and intended recipient, suspects the existence of the message. It includes the concealment of information within computer files. In digital steganography, electronic communications may include steganographic coding inside of a transport layer, such as a document file, image file, program or protocol.

For our purposes, we can use steganography to store binary data in web services that only allow image hosting. While researching the subject we stumbled across a unique property of the Graphics Interchange Format (GIF) and the Bitmap (BMP) image types [2, 10]. These image types, have a relatively simple byte arranging and the image's complete headers are on the top of the file. All data from a point on, are perceived as image-rendering data used by an image viewers to correctly render the image. After experimentation we found out that binary files could be added in images just by concatenating the new data at the end of the image and changing the size value of the images header. The images could still be read by image viewers, but they can also be used as vessels containing other files.

When we started uploading this kind of files, we noticed that all popular image galleries, converted our BMP formats to more compact JPG formats which effectively trashed our binary data. GIFs however are perceived as low-resolution files so no service tried to convert them. This handling of the GIF image type, enables a user (or in our case a parasitic storage system) to hide documents, music and all sorts of binary files in images and use online image galleries to store them. In addition to that, gif images are typically very small in size (a few Kilobytes) which means that there is very little wasted storage for each concealed binary file.

2.3.2 Cumulative File Transfers

Cumulative file transfers can be used when web services impose a time-wait penalty between successive downloads from their service. Using cumulative file transfers, the system can store and retrieve many files aggregated in one package instead of transferring each file atomically. This acts as a type of data prefetching, hiding the time-wait penalty from the user of the parasitic storage system. The downside of this method is that the system will have to re-upload the whole package of files every time a single file in it is edited. This overhead can be mitigated by: (a) lazily uploading the altered package, hoping that one upload can encapsulate more than one changes and (b) keeping the original file and uploading a separate *diff* file which will only contain the changes done to the original file.

2.3.3 Replication

Replication of files can also be helpful in combating the time-wait penalty policies. Instead of having a copy of a file on a single web service, the system can keep multiple copies of each file on different web services and access them in a round-robin fashion. This way, the system can switch between web services while it waits for the time-wait penalties to expire. The additional copies of each file can also be used in a disaster recovery scheme where one service deletes a file and the system restores it by accessing it through an alternate web service.

2.3.4 Account and Address Change

Most of the restrictive file policies use a combination of user accounts and IP address monitoring to enforce their limitations on available bandwidth and storage space. Both characteristics can be fooled by a determined user or system. More specifically, if a web service enforces a restrictive policy per account, the system can use multiple accounts and switch between them as needed. In addition to that, since most home Internet connections have dynamically assigned IP addresses, a simple router reboot can cause the assignment of a new IP address. Using this IP address, the parasitic storage system can commence a new download from a previously-restrictive web service, since the service assumes that a new user trying to access its resources.

2.4 Commercial Cloud Storage Solutions

In this section we briefly describe three commercial cloud storage solutions. We should point out that using a large pool of web services and data replication, our parasitic storage system can potentially provide service levels comparable to the following services, at zero cost for its users.

2.4.1 Amazon S3

Amazon S3 (Simple Storage Service) is an online storage web service offered by Amazon Web Services. Amazon S3 provides a web services interface that can be used to store and retrieve any amount of data, at any time, from anywhere on the web. Data are stored in *buckets* and retrieved via a unique, developer-assigned key. Amazon charges its users based on the amount of storage their objects use, the bandwidth they consume through the upload and download process and the requests made to their system. At the time of this writing, Amazon charges \$0.15 per Gigabyte of storage, \$0.17 per Gigabyte transfered and \$0.01 per one thousand `put,copy,post,list` requests.

2.4.2 Ubuntu One

Ubuntu One is a proprietary storage application and service operated by Canonical Ltd and currently in private beta. It enables users to store and sync files online and between computers. Ubuntu One has a client application that only runs on the latest version of Ubuntu, version 9.04. At the time of this writing, a free Ubuntu One account offers 2 Gigabytes of storage. A paid service is also available, which provides users with 10 Gigabytes of storage in exchange for a monthly fee of \$10.

2.4.3 Mandriva Click'n Backup

Mandriva Click'n Backup is a storage application which, as Ubuntu One, is designed to be integrated with linux distributions. Mandriva Click'n Backup doesn't offer a free account and its 20 Gigabyte storage plan costs $7.77.

3 Motivation

In this chapter we introduce the motives that lead us to this work, by highlighting the benefits of a complete parasitic storage solution.

3.1 Benefits of Parasitic Storage

The following benefits derive from the general concept of a parasitic storage system.

Zero Cost. The target of the parasitic storage system is to provide a service for free by taking advantage of the various online web services. Using free services, (such as Gmail, Picassa, Flickr, Rapidshare, etc.), a parasitic storage system can deliver tens of Gigabytes to an end user for zero cost.

Globally Accessible Storage. By default, web services can be accessed from any part of the Internet. This flexibility combined with our storage parasite provides globally accessible storage. A file that is stored under a remote web service through the parasitic storage system can be accessed by any host, anywhere in the world.

High Availability and Reliability. Existing web infrastructures go to great lengths in order to provide high availability and reliability of their services. Our storage parasite piggybacks on their effort and provides highly available and reliable storage to its users.

Filetype policies. Since parasitic storage relies on many online web services with unique characteristics and not on a local disk with uniform properties, the parasitic storage system can store each file on a service that provides the most flexibility for a particular filetype. For example, it can store a user's pictures on flickr because of the picture-driven API that flickr provides, or store a user's documents on Gmail to take advantage of the authorization step in Gmail; only the user providing the correct credentials to access the Gmail account, can in turn download her documents.

3.2 Benefits of ParasiticFS

The following benefits derive from the design choices and the resulting architecture of our parasitic storage system (explained in detail in the Design Section).

User-friendly File Namespace. Using ParasiticFS, a user can create as many directories as she likes, put files in the directory-depth of her own choosing, move around the files and generally be as flexible as she would be with a standard file system. All these operations are virtual, since the files are located on remote servers. Each time a user moves a file from directory A to directory B, all that is done, is moving the pointer to the particular remote file from directory A to directory B.

NFS-like semantics. Users can access their remote files as if the files were local similarly to other traditional technologies, like NFS. Moreover, using the services' global access, the user can manage her ParasiticFS files from anywhere in the world.

Disaster Recovery - While a user's data are stored on remote servers, the metadata of her files have to be accessed localy. ParasiticFS handles metadata consistency through the combined use of logging and checkpointing, both stored in a local disk device and periodically flushed to remote storage. The local disk copy of the log protects file system integrity against loss of volatile memory state while the remote log protects against more serious failures such as disk failures or site disasters. The frequency of flushing determines the recovery point in time.

Instant File Recovery - Since the files shown to the user, are in reality stored in remote servers all over the Internet, there is no overhead whatsoever in deleting a file. When a user chooses to delete a file, the storage parasite system hides the pointer to that file. Using the same mechanism, if a user deletes a file by accident, the system can recover the file in an instant, just by showing the ,previously hidden, link back to the user.

4
Design

In this chapter we describe our parasite's architecture and then we present in detail its main three components: the Remote Object Manager which uploads/downloads files from remote web services, the Parasitic Map which holds the bindings from local to remote files and the ParasiticFS, a user-level filesystem interacting with various web services through the Remote Object Manager.

4.1 Remote Object Manager

The Remote Object Manager (ROM) is the component which takes care of uploading and downloading the requested files to and from the various online web services. In our framework we view each web service as an object store which is responsible for storing and retrieving data. In Fig. 4.1 we depict Rapidshare and Gmail as two potential object stores. Each object store has

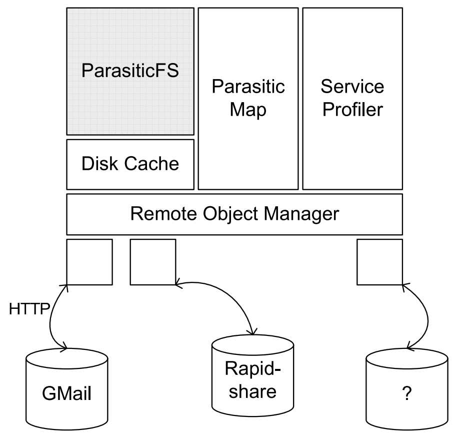

FIGURE 4.1: Architecture of the Parasitic Storage System

its own user-driven dialogues usually accessed through a web interface. Some object stores provide an API to the users who pay a subscription and no API to the free users, others provide an API with less functions to the free users and the rest provide no API at all. When an object upload or download is requested, ROM chooses the appropriate communication method (based on the object store requested) and handles the specifics of the transfer. Most object stores communicate over HTTP in which case ROM *crafts* the appropriate requests and orchestrates the upload/download process. When an object store provides an API, ROM uses its specifications to generate the requests. Alternatively, when an object store can only be used through the service's website, that is no free API provided, ROM simulates a browser visiting that website. The simulation is done by utilizing a valid User-Agent Header and following the steps (hyperlinks) that a real user would follow. ROM's modular architecture permits the addition of *drivers* for new object stores.

4.2 Parasitic Map

The Parasitic Map is the component which holds the translation from local files to remote files stored under each object store. This information is stored as records, where each record holds information about: (a) the file's local name, (b) file size, (c) the local path on the ParasiticFS, (d) the actual web service where it is stored and (e) various information on how to retrieve it (URLs, item identifiers, etc). This component is queried and updated by the ParasiticFS when a user interacts with the various remote files. In addition, the Parasitic Map is uploaded to an object store every time a user unmounts a ParasiticFS directory. This way, the various changes are populated to multiple mounting places. For example, a user can mount the filesystem at work, add some files, unmount it and mount it again later at her home. The ParasiticFS operating at her home will fetch the Parasitic Map from the specific object store and show the files that were added to the filesystem from another location (the user's workplace).

4.3 Service Profiler

The Service Profiler is responsible for maintaining the characteristics and constraints of the underlying object stores. It is used by the object allocator (when an object is created or a new version created through modification) to decide which stores can fulfill the `put` request. Similarly, the Profiler provides advice as to which store should be used to ensure a successful access at time t in the future. The problem of deciding where to allocate an object replica, or which store to retrieve a replica from, depends on the dimension (space, throughput, regulated bandwidth) one wants to optimize on.

4.4 ParasiticFS

The ParasiticFS component is a user-level file system which takes advantage of the storage parasite, by storing all of its files on object stores. It interacts with them through the Remote Object Manager and uses the Parasitic Map to translate local names and paths to remote web services and objects. More precisely, when a user mounts the ParasiticFS, an initialization script runs which reads the Parasitic Map and transform its records into filesystem nodes, directories and files. Every time a new file is added, an existing file is deleted, moved or renamed in the ParasiticFS the Parasitic Map is updated. Thus, the next time the user mounts the filesystem, her previous file operations are visible.

ParasiticFS works by wrapping all the filesystem-related system calls with its own implementation, interacting with remote object stores instead of the local storage medium. When a file is created or edited, the ParasiticFS uploads it to a remote object store using the ROM component. When a user requests access to a file, ROM uses information from the Parasitic Map to retrieve it from the proper object store. Once the file is downloaded, it is placed in a directory in the user's default filesystem. This directory acts as a *disk cache*, which serves requests to previously accessed files.

All of these actions are transparent to the user or an application accessing the filesystem. It is important to point out that no data is exchanged with remote object stores, when a file is copied, moved, renamed or deleted in the local ParasiticFS. The above actions are completed instantly, giving the illusion that the files are actually on the user's local disk yet they do not consume any storage space. It also provides the familiar directory abstraction in which the user can create directories and group her files together in some meaningful way (music/ , pictures/ , videos/ etc.). These directories are "virtual" and they exist only while the ParasiticFS is running.

Disk Cache. A disk cache is used in the ParasiticFS to speedup file accessing. The disk cache is a directory in the user's default filesystem in which files that were previously downloaded or uploaded by the ParasiticFS, exist. While this cache could be limited by the actual disk capacity of the user's hard drive, ParasiticFS considers it a limited-size cache (practically a few Gigabytes) and periodically frees up space consumed by the least recently used files stored under it.

5
Implementation

In this chapter we discuss the implementation details of our prototype. Using Python [16] as our language of choice we managed to write a compact implementation of our system's prototype: the Remote Object Manager, the Parasitic Map and the ParasiticFS source code combined are 700 lines long.

5.0.1 Remote Object Manager

Recall from Section 4 that ROM is the component responsible for the uploads / downloads to and from the various web services. We have currently implemented drivers supporting data transfers on four different object stores, namely Rapidshare, Sendspace, DepositFiles and GMail [6,11,22,32]. Rapidshare, Sendspace and Depositfiles are designed to support free file hosting while GMail supports mailing operations. We chose these services because they have a large user base, implying high reliability and availability. Rapidshare and Depositfiles can only be used through their web pages while Sendspace and GMail provide an API [12, 23]. Using Python and the `libcurl` [15] library for sending and manipulating HTTP requests we fetch the appropriate pages and fill the various forms in order to simulate the actions done by a user.

The following code excerpt is from the upload functionality of the Sendspace object store driver.

```
def uploadSSpace(self, filename):
        print "Sendspace Upload: \n"

        #Regular expression used to find the correct token credential
        page = self.getURL("http://api.sendspace.com/rest/?method=
                                    auth.createtoken&api_key=WOFZIWYF9C&
                                api_version=1.0&response_format=xml
                                &app_version=0.1")
        p1 = re.compile("<token>(.*)<\/token>")
        m =  re.search(p1,page)

        [..]

        tokened_passwd = md5.new(token + passMd5).hexdigest().lower()

        #Get session id
        page = self.getURL("http://api.sendspace.com/rest/?method=auth.
                                login&token=" + token + "&user_name="
                                + username + "&tokened_password=" +
                                tokened_passwd);
        [..]

        #Get upload info
        page = self.getURL("http://api.sendspace.com/rest/?method=
                                upload.getinfo&session_key=" +
                                session_key +"&speed_limit=0");

        #Initiate the upload using the correct credentials extracted
        #from previous steps

        c = pycurl.Curl()

        pf = [        ( 'MAX_FILE_SIZE', '314572800'),
                ( 'UPLOAD_IDENTIFIER', upid),
                ( 'extra_info', exinfo),
                      ( 'userfile', (c.FORM_FILE, filename))
        ]

        c.setopt(c.URL, formURL)
        c.setopt(c.USERAGENT,"Mozilla/5.0 (Windows; U; Windows NT
                        5.1; en-US; rv:1.8.1.1) Gecko/20061204
                        Firefox/2.0.0.1")
        [..]


        #Return the download link and remove link to the caller
        return [dLink,rLink]
```

5.0.2 Parasitic Map

Recall from section Section 4 that the Parasitic Map is the component which holds the mappings from local files to remote files on web services. We chose to implement this component as an ASCII text file, holding information in the form of records. Our choice was mainly driven in favor of rapid prototyping. However, it is trivial to implement the Parasitic Map component using more appropriate technologies like SQLite and MySQL [18, 24]. In Table 5.1 we list the first few lines of our working prototype's Parasitic Map. In favor of clarity, we have ommitted the last *accessed*, last *changed* timestamps, and the *delete-file* hyperlink provided by the file hosting object stores.

Path	Filename	Size	Timestamp	Download URL
/	foo.txt	460	1241262894	www.sendspace.com/file/lgxcjg
/	wafl.pdf	72451	1241254109	www.sendspace.com/file/zsmqqv
/	dog1.jpg	1567246	1243508778	www.sendspace.com/file/gd8blv
/	50m	52428800	1242903879	www.sendspace.com/file/oof4ua

Table 5.1: Information held in the Parasitic Map component

5.0.3 ParasiticFS

Recall from section Section 4 that the ParasiticFS is an implementation of a user-level filesystem which enables users and applications to interact with remote files as if they were stored locally. We chose to implement ParasiticFS using the FUSE library [8, 29]. For each filesystem-related system call we provide our own functions.

For example, when a user creates or copies a new file in the ParasiticFS, `create()` is called. Inside `create()` we have placed code to mark the file as *new*. When in turn, `release()` is invoked for that particular file, ROM is used to upload the file to the appropriate object store. In addition, the local Parasitic Map is updated to include metadata for the new file. In a similar fashion, when `write()` is invoked for an existing file, ParasiticFS marks the file as *modified* and re-uploads it upon a `release()` operation in-order to be consistent. Finally when `open()` is invoked, the ParasiticFS checks its local disk cache for presense of the requested file. If the file is not found, a request is forwarded to ROM for that particular file. The file is fetched in the local disk cache and any subsequent reads to that file are served by the disk cache. The following two code excerpts are taken from the implementation of the `open()` and `release()` system calls, demonstrating the communication of ParasiticFS with the Remote Object Manager component.

```
def open(self, path, flags):
        self.fd += 1

        #Find the download url associated with the file being accessed
        durl  = self.filedict[path]["durl"]

        [...]

        #Start download if the file isn't in the disk cache
        if not os.path.exists(cache_dir + path):
                self.timecost[path]["net_start"] = time()
                self.parasitic.download(object_store, durl,cache_dir
                                        + path )
                self.timecost[path]["net_end"] = time()

        self.openfiles[path] = os.open(cache_dir + path, os.O_RDWR)
        return self.fd
```

```
 def release(self, path, fh):

        #Upload file if it is new or modified
        if (path in self.newfiles) or (path in self.mod_files):

                #Upload the requested file
                data = self.parasitic.upload(object_store, cache_dir + path)

                #Extract correct folder information
                new_folder = path.rpartition("/")
                if new_folder[0] == "":
                        folder = "/"
                else:
                        folder = new_folder[0]

                #Add record to Parasitic Map structure
                self.filedict[path] = ({"folder":folder, "fname":new_folder[2],
                                        "fsize":self.files[path]['st_size'],
                                        "ctime":int(time()), "mtime":int(time()),
                                        "atime":int(time()), "durl":data[0],
                                        "rurl":data[1]})

                [..]
                if path in self.newfiles:
                        os.close(self.newfiles[path]);
                        del(self.newfiles[path])
                [..]

        #File is not new and it is not modified
        else:
                os.close(self.openfiles[path])
                del(self.openfiles[path])
        return 0
```

6 Evaluation

In this chapter we explore the performance of our resulting parasitic storage system and the availability of remote object stores. We distinguish the performance of our system in network performance and local performance. In network performance we measure the available upload and download bandwidth of four services in a period of a day and the total aggregated storage that we managed to access. In local performance, we are more interested in measuring the overhead of ParasiticFS compared to a native Ext3FS by comparing the `read` and `write` speeds of our system.

6.1 Bandwidth and Storage

In the first experiment we uploaded and downloaded 12 Gigabytes, splitted as 60 files of 200 Mbytes each to three different file-hosting services using ParasiticFS. We use a round-robin access policy to avoid access-frequency

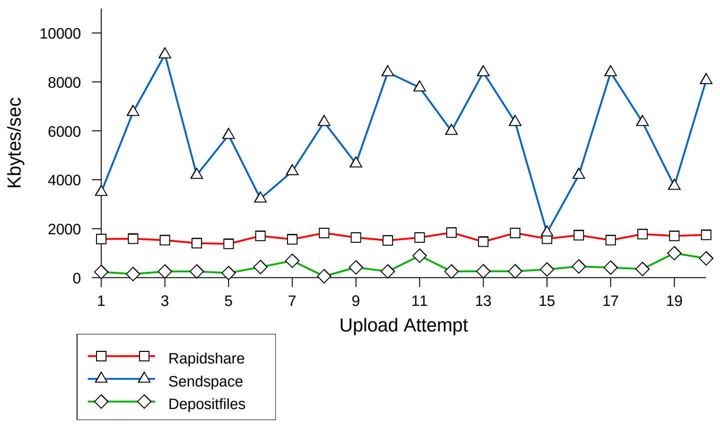

FIGURE 6.1: Average upload rate for 60 uploads of 200Mbyte files on 3 different file-hosting services

limits put in place by two of the object stores (Rapidshare, Depositfiles). By measuring the average upload and download speed for each transfer - Fig. 6.1,6.2 - we found out that (a) web-services are willing to accept a large load of data in a short amount of time (b) the average download and upload speed of each transfer is greater or at least equal to the maximum download / upload rate of an average home user's Internet connection and (c) object-stores (web services) favor creation of new content over access to existing data (i.e., uploads are faster than downloads).

In addition to the 12 Gigabytes uploaded in three file-hosting services, we also transfered and measured 20 files of 10 Mbytes each to a GMail account, Fig. 6.3. GMail allows a maximum attachment size of 10 Mbytes which is not very convinient for storing large files, however the *Static Data Location* and *Implicit File Permissions* properties that it has (see Section 2), make it a great target for our system.

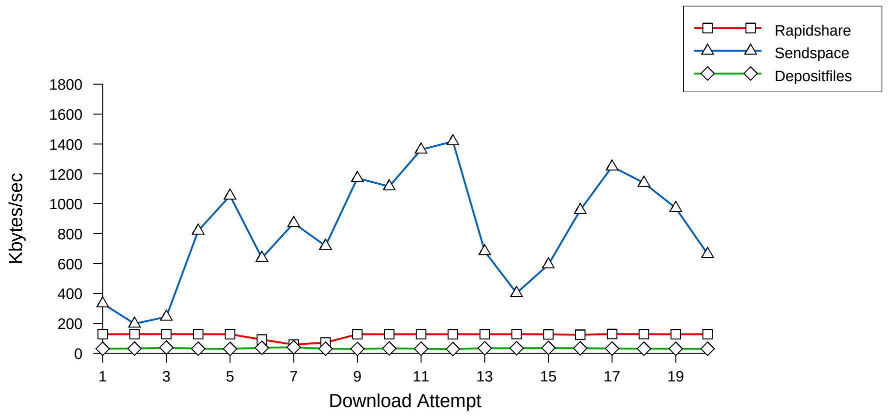

Figure 6.2: Average download rate for 60 downloads of 200Mbyte files from 3 different file-hosting services

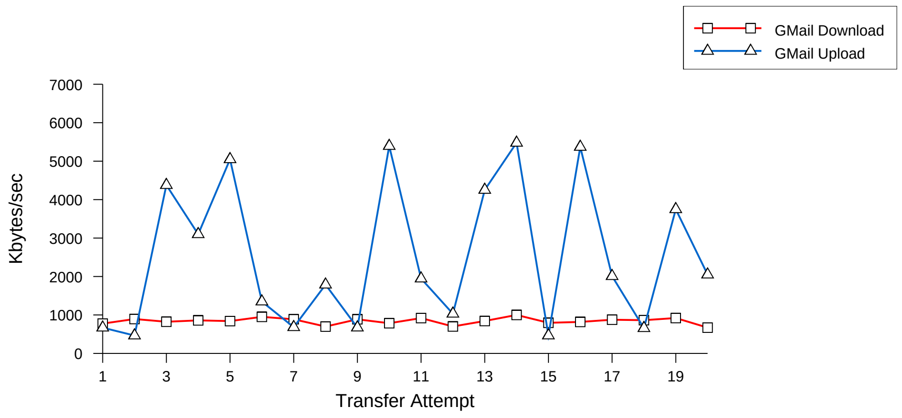

Figure 6.3: Average upload/download rate for 20 files of 10 Mbytes each, transfered to and from GMail

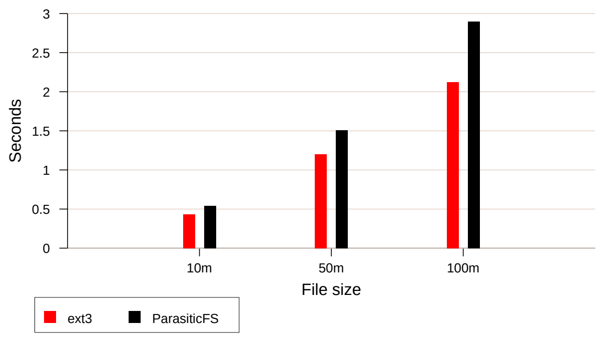

FIGURE 6.4: Comparison of seconds needed by md5sum to compute checksum for files of different filesizes

6.2 ParasiticFS Performance

6.2.1 Read

In the third experiment we quantify the ParasiticFS `read` speed compared to a native Ext3FS. We measure the time needed by the *md5sum* application to compute a checksum for 3 files of different filesizes when the file is stored in an Ext3FS partition and when it is stored in the ParasiticFS partition and present in the filesystem's disk cache - Fig. 6.4. While the overhead of the ParasiticFS writes, ranges between 25-35 % we should note that this delay is caused by the implementation of our prototype rather than the concept of a parasitic filesystem. Our prototype intercepts each system call in user-space and if necessary forwards it to kernel-space. In this experiment, all reads are forwarded to kernel-space reads since the files are present in the disk cache. This user-level mediator-behavior needs an extra memory copy for every `read()` which in turn causes the delay. If our prototype was implemented directly in kernel-level, the extra memory copy wouldn't be needed thus the performance of cache-hits would be comparable to native filesystems.

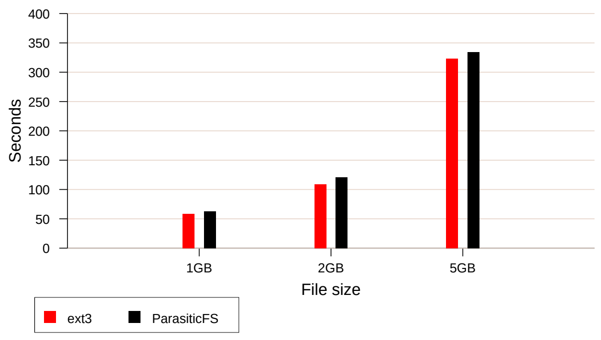

FIGURE 6.5: Comparison of seconds needed by cp to copy files of different filesizes

6.2.2 Write

In the fourth experiment we quantify the ParasiticFS `write` speed compared to a native Ext3FS. We measure the time needed by the *copy* system command in order to copy three different files, each with a different file size. More precisely, we measure the time needed for a file copy from an Ext3FS partition to an Ext3FS partition and from an Ext3FS partition to a ParasiticFS partition. While the copy application needs a `read()` for every `write()`, the total number of reads and the source filesystem are the same in both cases, thus the comparison between Ext3FS and ParasiticFS `write()` system call is valid. In Fig. 6.5 we can see that the overhead of the ParasiticFS reads is about 10 % which is really small compared to the read overhead, measured in the previous experiment. Tha ParasiticFS implementation still acts as a user-level mediator for the `write()` system call but in this case, no data have to be returned back to the user, thus no extra memory copy occurs. This experiment clearly shows that the (write) performance of ParasiticFS is comparable to the performance of native filesystems.

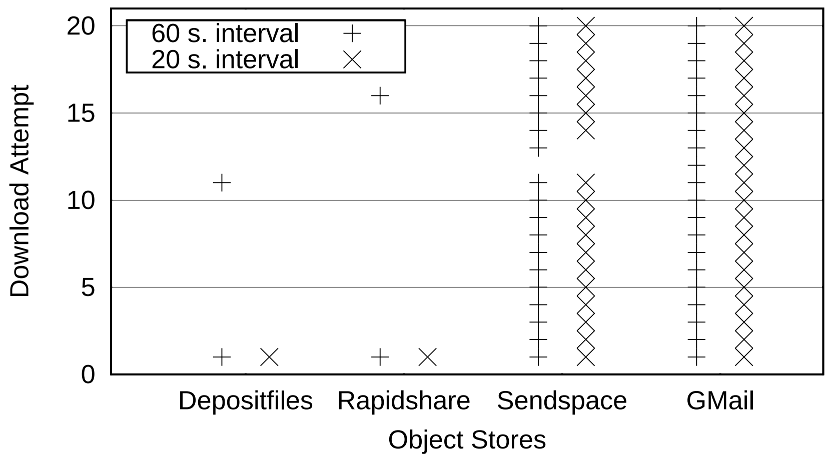

FIGURE 6.6: Object-Store Availability.

6.2.3 Object-store Availability

In this experiment we replace our round-robin access policy by a policy of successive accesses to each service. Our observed metric (in the y-axis of Figure 6.6) is the success or failure of each individual download in a sequence of 20 attempts to download a 10MB file. We run the experiment with two different inter-access delays: 60s and 20s (left and right column for each object store in Figure 6.6). Each data point in Figure 6.6 signifies a successful download attempt.

A key observation is that file-hosting services have a download-frequency limit for non-paying users (15min for Rapidshare, 10min for Depositfiles) which prohibits them from downloading content until the inter-access time limit expires (uploads are unregulated). This indicates that parasitic storage systems cannot rely solely on any single such system for general use but must multiplex several file-hosting services and combine them with other types of persistent-content services. Figure 6.6 shows that GMail and Sendspace are more lenient and do not impose significant frequent-access limitations (at least none observable at the time scales used in this study). Sendspace however, seems to activate a bandwidth-restriction mechanism after the 11th successive download, which vastly limits download bandwidth for subsequent transfers. It is important to note that our reluctance to try more frequent access patterns on GMail and Sendspace for fear of being shut out of service during this study highlights the fine balance a parasitic storage system must walk.

7 Limitations and Future Work

So far we have explored the need for network storage, the commercial solutions storing data *in the cloud* and how a storage parasite taking advantage of the aggregated free storage space of existing web services can provide cloud storage for zero-cost. We understand that the key limitation of our storage parasite system is the limitation all network storage solutions have. All un-cached file operations will have a maximum throughput equal to the user's network throughput. While home network throughput can be as low as a few hundred Kbits in developed countries, it is widely accepted that networks are becoming faster each year [4]. Thus, we argue that the users who may not be able to access quickly enough their remote files today, will probably be able to do so in a year from now.

Although most of the well-known web services have high availability and reliability, our storage parasite currently provides no guarantees that data stored in the various remote web services will not be lost since none of the utilized services make such promises. We have explored the solution of replicating data on distinct web services and we are considering the use of selective file replication in-order to ensure that sensitive files such as documents, source code files and images will not be lost if a single service fails. We want to avoid replicating each and every file since the user could store downloaded movies and games in the parasitic storage system which are both large in size yet easily recovered in case of a data loss.

While we have implemented drivers for four different object stores (Rapidshare, Sendspace, Gmail, Depositfiles), our parasitic storage system only uses the Sendspace driver for uploading and downloading files. Sendspace is used because it has the least limitations and better performance compared to the other object stores. A seperate component is needed, which will balance the use of all available object stores and keep track of all their usage and their limitations.

On the filesystem part of our work, we realize that some core-parts of a complete network filesystem are missing. ParasiticFS does not implement a locking service and it is currently a single user file system since we don't support any cache coherence mechanism between simultaneous mounts of the same filesystem. We are currently investigating several centralized and distributed approaches to the above problems and we are confident that we will be able to present complete and novel solutions to them in the near future.

8
Related Work

The term “parasitic computing” first appeared in Nature Magazine in 2001 [1] where the authors forced Internet servers to solve a piece of a complex computational problem, namely an NP-complete problem, by engaging them in standard protocol communication. The checksum function of the TCP packets was used to perform the desired computations while target servers where unaware that they participated in such a scheme.

Kiselyov presented HTTPFS [14] in 1999. HTTPFS is a user-space filesystem that can access and modify remote files over HTTP. The main difference from our work, is the fact that HTTPFS needs remote storage servers to run a specific perl cgi-script. This means that the user must be in control of at least one remote server, whereas our system uses existing web services parasitically eliminating the need for control of a remote storage server. In addition, parasitic storage presents file-data from multiple web services in a uniform namespace instead of data from just one remote server.

In 2006, Traeger et al. [25] used search-engine caches and mail services for backing up their files. The authors used scripts which executed when they wanted to backup or recover specific files. While we too use mail services as one type of service that can be used for parasitic storage, we use a file-system approach where all remote web services are "mounted" on a user's operating system, providing transparent download/upload functions instead of user-executing scripts. This approach enables the use of parasitic storage as a media filesystem, or an NFS-like service in addition to a backup service.

In [20, 21] the authors used the idea of delayed communication channels to store parasitic data on the wire, by constantly exchanging it with remote hosts. They did that by sending and constantly replaying data packets using the ICMP echo service and presenting this data using a uniform user interface. While this is an attractive idea for sensitive files that a user doesn't want to store on a disk medium (in fear of recovery after their deletion), it is inpractical for stable storage since the moment that a user stops juggling his data - e.g. the user's computer shutting down - that moment all data are lost. The user cannot use her data from any other host and considering that the average home user has a limited upload bandwidth, the maximum capacity of the data being successfully stored on the wire are much less than the potential storage provided through the parasitic use of remote web services.

Our system treats remote web services such as GMail or Rapidshare, as object stores which only export a `put` and `get` interface. The complexity of efficiently storing and accessing data is left to each service's internal infrastructure. This approach is similar to the "Network Attached Secure Disks" (NASD) [9] approach were commodity storage components encapsulate additional functionalities such as storage self-management and cryptographic support.

Filesystems that take advantage of a specific web service, such as GmailFS [13], TwatFS [5] and Graffiti Networks [19], have emerged the last couple of years. GMailFS stores files as emails on specific gmail accounts, TwatFS stores files as text in Tweeter [26] and Graffiti Networks encodes binary data as text and stores it in MediaWiki pages [17]. All of these filesystems act "parasitically" and store binary data in web services that where not designed for file storage but at the same time they are service specific. On the other hand, our system works collectively by aggregating multiple distinct web services under a new storage service. Instead of having multiple service-specific filesystems, we use web-service specific drivers and a single filesystem where each file can be stored on a different web service.

9
Conclusion

A key finding of our study is the diversity of behavior among persistent-content Web services. The four services we evaluated can be roughly categorized into those that provide limited storage capacity (typically a few GBs per user), high bandwidth, lenient access-policies, and reliable storage of user content (we refer to this class as *first-class* storage); and those that provide abundant storage capacity but limited access bandwidth, severely limited access-policies, and reduced user-data reliability (*second-class* storage). GMail, Rapidshare, and Depositfiles seem to fit this categorization. Sendspace on the other hand seems to fall somewhere in between. We expect this diversity and complexity in behavior to increase in the future.

It is vitally important for any parasitic storage system to maintain accurate profiles for the underlying services (e.g. using benchmarks such as those used in our experiments) and best utilize them according to their distinct characteristics, policies and limitations to meet application needs. A parasitic storage system should combine first- and second-class storage (as well as other classes that will undoubtedly form as services evolve), as appropriate for different types of user data.

In this thesis we presented a new storage system called Parasitic Storage. Parasitic Storage is a storage parasite which uses the free online disk capacity provided by online web services in order to create a new free storage service. Instead of viewing online web services as isolated entities providing certain functionalities, Parasitic Storage treats them as connected components of its storage service. Observing the details of each web service we were able to see past their individual behavior (custom web interfaces, different steps needed to perform a common functionality) and deduce a common upload/download functionality in all services used. Through this observation we implemented a generic API that transcended each service's specific mechanisms. Using that API we implemented a user-level filesystem called ParasiticFS.

ParasiticFS, is a file system which presents all remote files as files stored on a local storage medium by mounting distinct web services under a unified namespace. Users are able to create files, copy them, move them and organize them in directories as they would do with a regular filesystem while their actions are transparently mapped to local and remote operations. In addition, due to the familiar filesystem semantics, applications that work with local files need no modification to work with ParasiticFS.

Using our ParasiticFS prototype to access aggregated storage we showed that major web services do not place limits on storing large amounts of data in short periods of time, matching our desired requirements and enabling the use of ParasiticFS as a backup, media or all-purpose network storage filesystem. We are currently working on extending our analysis and demonstrating the viability of parasitic storage and its trade-offs for real applications.

Bibliography

[1] A.-L. Barabasi, V. W. Freeh, H. Jeong, and J. B. Brockman. Parasitic computing. *NATURE*, 412:894–897, August 2001.

[2] Rfc797 - format for bitmap files. `http://www.faqs.org/rfcs/rfc797.html`.

[3] I. A. Bo, B. Hong, E. L. Miller, S. A. Brandt, and D. D. E. Long. Managing flash crowds on the internet. In *In 11th Modeling, Anal. & Simulation of Comput. & Telecomm. Syst*, pages 246–249, 2003.

[4] P. Chanclou, Z. Belfqih, and B. C. et al. Access network evolution: optical fibre to the subscribers and impact on the metropolitan and home networks. *Comptes Rendus Physique*, 9(9-10):935 – 946, 2008. Recent advances in optical telecommunications.

[5] CP, Adam, Frank$\hat{2}$, and Vyrus. Twatfs: A surly abuse of social networking. In *LayerOne*, 2009.

[6] Deposit files. `http://www.depositfiles.com/`.

[7] Biggest list of free file hosting services updated monthly. `http://www.110mb.com/forum/general-chat/biggest-list-of-free-file-hosting-sites-updated-monthly-t1428.0.html`.

[8] Fuse:filesystem in userspace. `http://fuse.sourceforge.net/`.

[9] G. A. Gibson, D. F. Naglet, K. Amirit, J. Butler, F. W. Chang, H. Gobioff, C. Hardint, E. Riedelf, D. Rochberg, and J. Zelenka. A cost-effective, high-bandwidth storage architecture. In *In Proceedings of the 8th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 92–103, 1998.

[10] Gif graphics interchange format, version 89a. `http://www.digitalpreservation.gov/formats/fdd/fdd000133.shtml`.

[11] Gmail: Email from google. `http://mail.google.com/`.

[12] libgmail - python binding for google's gmail service. `http://libgmail.sourceforge.net/`.

[13] R. Jones. Gmail filesystem - gmailfs. `http://richard.jones.name/google-hacks/gmail-filesystem/gmail-filesystem.html`.

[14] O. Kiselyov. A network file system over http: remote access and modification of files and *files*, 1999.

[15] curl and libcurl. `http://curl.haxx.se/`.

[16] M. Lutz. *Programming Python*. O'Reilly Media, Inc., 2006.

[17] Mediawiki. `http://www.mediawiki.org`.

[18] Mysql. `http://www.mysql.com/`.

[19] A. Pavlo. Graffiti networks project: A subversive, internet-scale file sharing model. `http://graffiti.cs.brown.edu/`.

[20] W. Purczynski and M. Zalewski. Juggling with packets: floating data storage. `http://lcamtuf.coredump.cx/juggling_with_packets.txt`.

[21] K. Rosenfeld, H. Sencar, and N. Memon. Volleystore: A parasitic storage framework. In *Information Assurance and Security Workshop, 2007. IAW '07. IEEE SMC*, pages 67–75, June 2007.

[22] Sendspace: Send files the easy way. http://www.sendspace.com/.

[23] Sendspace.com: Api how to. http://www.sendspace.com/dev_howto.html.

[24] Sqlite. http://www.sqlite.org/.

[25] A. Traeger, N. Joukov, J. Sipek, and E. Zadok. Using free web storage for data backup. In *StorageSS '06: Proceedings of the second ACM workshop on Storage security and survivability*, pages 73–78, New York, NY, USA, 2006. ACM.

[26] Twitter: What are you doing? http://twitter.com/.

[27] How much space do you use with gmail? http://forums.macrumors.com/archive/index.php/t-588324.html.

[28] Gmail users: How much space are you using? http://www.answerbag.com/q_view/930394.

[29] G. Verigakis. fusepy: Python bindings for fuse with ctypes. http://code.google.com/p/fusepy/.

[30] 3 billion! flickr blog. http://blog.flickr.net/en/2008/11/03/3-billion/.

[31] Wikipedia: Gmail. http://en.wikipedia.org/wiki/Gmail.

[32] Rapidshare: Easy file hosting. http://rapidshare.com/wiruberuns.html.