A Cloud Storage Overlay to Aggregate Heterogeneous Cloud Services

Guilherme Sperb Machado, Thomas Bocek, Michael Ammann, Burkhard Stiller
Department of Informatics IFI, Communication Systems Group CSG, University of Zurich
Binzmühlestrasse 14, CH-8050 Zurich, Switzerland
[machado|bocek|stiller]@ifi.uzh.ch, michael.ammann@uzh.ch

***Abstract*—Many Cloud services provide generic (*e.g.*, Amazon S3 or Dropbox) or data-specific Cloud storage (*e.g.*, Google Picasa or SoundCloud). Although both Cloud storage service types have the data storage in common, they present heterogeneous characteristics: different interfaces, accounting and charging schemes, privacy and security levels, functionality and, among the data-specific Cloud storage services, different data type restrictions. This paper proposes PiCsMu (Platform-independent Cloud Storage System for Multiple Usage), a novel approach exploiting heterogeneous data storage of different Cloud services by building a Cloud storage overlay, which aggregates multiple Cloud storage services, provides enhanced privacy, and offers a distributed file sharing system. As opposed to P2P file sharing, where data and indices are stored on peers, PiCsMu uses Cloud storage systems for data storage, while maintaining a distributed index. The main contribution of this work is to show the feasibility to store arbitrary data in different Cloud services for private use and/or for file sharing. Furthermore, the evaluation of the prototype confirms the scalability with respect to different file sizes and also shows that a moderate overhead in terms of storage and processing time is required.**

***Index Terms*—Cloud Computing, Cloud Services, Cloud Storage, Cloud Overlay, Peer-to-peer, DHT, Data Validation**

I. Introduction

A wide variety of Cloud Services (CS) are available today, such as Amazon EC2 [1], SkyDrive [14], Google App Engine [5], or Dropbox [4]. These CSs can be categorized into 3 types: Infrastructure-as-a-Service (IaaS), Platform-as-a-Service (PaaS), and Software-as-a-Service (SaaS). In all of these three categories, there exist CSs for storage purposes, to provide data storage through a well-defined Application Programming Interface (API). On one hand, services supporting any data types are termed *generic* Cloud storage services. On the other hand, there exist CSs that support storage of a restricted set of data types. Those are termed *data-specific* Cloud storage services. Typically, CSs offering data-specific storage employ a data validation scheme of certain data types and/or properties. *E.g.*, Google Picasa provides an API, where users are able to publish and organize pictures and albums, allowing the upload of images with a certain file format (such as JPG, PNG, or BMP), resolution, and size.

Both types of Cloud services — with generic and data-specific storage — show one common aspect: data is stored on CS's servers. However, both types of CSs are still heterogeneous: they offer different APIs, different accounting and charging schemes, different privacy and security levels, different functionality, and, among generic Cloud storage services, they present different data type restrictions. Therefore, the heterogeneity between CSs turns the task of aggregating CSs' storage (into one single storage entity) a challenging task. Aggregating multiple CSs allow end-users to have more integrated storage space, a single entry point to store data, and, depending on how data is managed among CSs, more data reliability and privacy.

This paper addresses the following research questions: (1) Is it possible to build an overlay (*i.e.*, a network on top of another network) that explores generic *and* data-specific storage of CSs in order to store, retrieve, and share any kind of files? (2) Would such an overlay scale with respect to different files (with different sizes) being stored, retrieved, and shared? (3) How much overhead is required to exploit jointly generic and data-specific storage of CSs? Hence, this paper presents and evaluates a new approach of a Cloud storage overlay, entitled Platform-independent Cloud Storage System for Multiple Usage (PiCsMu) [17]. PiCsMu stores data on heterogeneous CSs independent of its Cloud category, *i.e.*, IaaS, PaaS, or SaaS, (a) by aggregating multiple CSs' data storage capabilities to be seen as one single storage entity, (b) by providing enhanced privacy levels, and (c) by enabling a distributed file sharing network relying on CSs' storage instead of peers' storage. Even though the aggregation of CSs' storage within an overlay may loose specific CSs' features (like, *e.g.*, rotating a picture within Google Picasa or version control within Dropbox), PiCsMu brings specific storage advantages as (a), (b), and (c).

PiCsMu employs the following 4 steps to aggregate CSs' storage and upload files: (1) fragmentation, (2) encryption, (3) data encoding, and (4) data upload to CSs. Each of these steps requires a minimal overhead with respect to processing time and storage. These steps, combined, add another layer of tasks to reconstruct original files, therefore turning it even harder for an attacker (or, in case of data leakage, non-authorized users) to gain access to the stored content. The fragmentation step brings advantages in data reliability and redundancy, since multiple fragments can be stored in multiple CSs, thus, preventing data loss if a single CS shuts down its services (*e.g.*, Megaupload [21]) or if a CS presents serious availability problems.

Since the resulting information related to all 4 steps forms the PiCsMu Index, which represents what, how, and where data was stored, file sharing functionality effects scalability. PiCsMu uses a hybrid mechanism that can store this index either centrally, for private storage, or in a Peer-to-peer (P2P) network enabled by a Distributed Hash Table (DHT), for sharing purposes. The evaluation focuses on measurements related to the file upload and download times, scalability considering different file sizes, and system overhead. Although PiCsMu supports public CSs, the evaluation uses a local CS within a

local network in order to perform measurements considering the best case scenario (excluding external factors, *e.g.*, delays, which are not part of PiCsMu).

The remainder of this paper is organized as follows. Section. II outlines basic terminology and related work, which is followed by the PiCsMu System Architecture and Design in Section. III. The PiCsMu Prototype (Section. IV) is discussed and was used to perform evaluations (Section. V). Finally Section. VI summarizes the work and addresses future work.

II. Terminology and Related Work

An *overlay* network (or overlay in short) is a virtual or logical network on top of another network with addressable endpoints [12]. Overlays are often used to provide a routing topology not available in the underlying network. *Encoding* is the process in which data is converted into another form. Possible encoding applications include reduction of the file size (*e.g.*, compression) or hiding data inside other file formats to conceal the original content (*e.g.*, steganography) [9]. Decoding is the reverse process to restore the encoded data to their original form. A *credential* is the attestation of authority to access a given *CS account*. This can be achieved through providing a security handle, *e.g.*, an OAuth [15] token with restricted access (time out) or a username/password pair with full access to that account.

TABLE I: A Feature Comparison of Cloud Storage Services.

Feature	Generic Cloud storage services	SpiderOak	Wuala	Otixo	PiCsMu
Overlay	-	-	-	✓	✓
Additional Service Support	-	-	-	✓	✓
Fragmentation to Multiple Clouds	-	-	-	-	✓
Built-in Client-side Encryption	-	✓	✓	-[a]	✓
Encoding	-	-	-	-	✓
Decentralized Index	-	-	-	-	✓

[a]Encryption might be offered by underlaying CSs only.

In order to compare related work to PiCsMu, selected related work is divided into two groups: first, a comparison to generic Cloud storage services and, second, to P2P file sharing systems. Amazon S3 [1], Dropbox [4], Google Drive [5], SkyDrive [14] are examples of generic Cloud storage services. SpiderOak [19] and Wuala [22] have been selected, since they are known for Cloud storage services with encryption on the client-side. Otixo [16] was chosen, because it presents an overlay and supports multiple Cloud services. The comparison of related generic Cloud storage services in Table I presents six key features, "✓" describing the presence of the specified feature, while "-" denotes the lack of it: "Overlay" defines that the CS builds a management network on top of other CSs. Therefore, the CS does not store content data itself, but in an underlaying CS. "Additional Service Support" represents whether a CS supports the aggregation of other CSs' storage, thus, expanding its storage capabilities. "Fragmentation" indicates whether a CS can split up the file and store it with other CSs, thus, fragmenting it into multiple Clouds. "Built-in Client-side Encryption", refers to the encryption done by the end-user application, rather than manually by the end-user or by the CS system. "Encoding" indicates whether data can be transformed to a well-known data type that is accepted by restricted Cloud storage services. Finally, "Decentralized Index" indicates whether the index of files is stored in a distributed manner.

PiCsMu and Otixo offer Cloud storage overlay services. The advantages of using an overlay is that data is stored in third-party servers, and the overlay system can decide where to store based on policies (storage management). Moreover, users need to interact with the overlay system only, instead of accessing each service itself. The fragmentation enables to use many Cloud storage services, giving the user a bigger size of storage and providing data redundancy. Only SpiderOak, Wuala, and PiCsMu use client-side encryption, but the other Cloud storage services may use a service that provides it. Using server-side encryption, the service provider holds the encryption and decryption key pair. However, pure server-side encryption allows the provider to view and access all files and thus, only provides privacy in case of servers are compromised without compromising the keys. PiCsMu offers built-in client-side encryption, where all data is encrypted on the end-user machine before sending it to the CS. The disadvantage is that if the user loses the password, the data cannot be accessed anymore. The encoding provides the possibility to store data in any Cloud storage service, even if the Cloud storage service uses data validation just accepting files of specific file formats (*e.g.*, SoundCloud accepts audio files only). Although Otixo is the closest approach to PiCsMu, to the authors best knowledge, PiCsMu is the only system known to fragment, encrypt, and encode files in order to distribute fragments in multiple CSs. Machado *et al.* [13] already investigated how to bypass the data validation process of CSs to store arbitrary data without restrictions, where encoders were implemented in an early version of the PiCsMu System.

TABLE II: A Comparison of P2P File Sharing Systems.

Criteria	Napster	Gnutella	BitTorrent	FreeNet	PiCsMu
Topology	Central-ized	Decen-tralized	Central-ized	Decen-tralized	Decen-tralized
Architec-ture	Unstruc-tured	Unstruc-tured	Unstruc-tured	Unstruc-tured	Struc-tured
Lookup	Central Index	Flooding	Tracker/ DHT	Key-based	DHT
Storage on Peers	✓	✓	✓	✓	-
File Search	Internal	Internal	External	External	Internal
Download	Peers	Peers	Peers	Peers	CSs
Upload	Peers	Peers	Peers	Peers	CSs
Private Sharing	-	-	✓	-	✓

The PiCsMu System presents a share functionality, which uses a P2P network. Thus, the comparison of related P2P file sharing systems is presented in Table II, where the following dimensions are taken into consideration: "Topology" determines whether the network topology is centralized or decentralized, "Architecture" illustrates the overlay scheme as structured or unstructured, "Lookup" represents the implemented protocol as being able to query other peers for infor-

mation, “Storage on Peers” describes whether peers store file data, “File Search” determines how the user can search for files within the P2P network, “Download” determines the entity, where data is downloaded from, “Upload” determines the entity, where data is uploaded to, “Private Sharing” determines if the system supports closed groups of peers.

PiCsMu uses a DHT, which is a common type of structured overlay and guarantees a lookup time of O(log N), where N is the number of peers. Queries in structured overlays are more efficient, in contrast to queries in most unstructured overlays [12]. Although BitTorrent and PiCsMu are systems that are based on a structured overlay for storing information about files (*i.e.*, file metadata), PiCsMu differs in the sense of file storage locations: While in BitTorrent fragments are also stored on peers, PiCsMu fragments are stored within CSs. Previous work [11] shows a problem in P2P networks: massive content distribution is often disrupted or suffers from poor performance by churning peers. Thus, CSs are considered stable systems compared to peers within a P2P network. Addressing the issue presented in [11], PiCsMu stores information about files on peers (index), which represents much less data than the file, thus having a faster replication to more peers to counter the effects of churn.

III. PiCsMu System Architecture and Design

PiCsMu defines how and where files are handled/stored. Thus, the designed file upload and download processes for storing and retrieving files within the PiCsMu System is explained, presenting three modes of operation: *private storage*, *private sharing,* and *public sharing*. While private storage relies on a centralized index, private and public sharing modes are used with the support of a P2P network. Finally, all three modes of operation are presented in a use case.

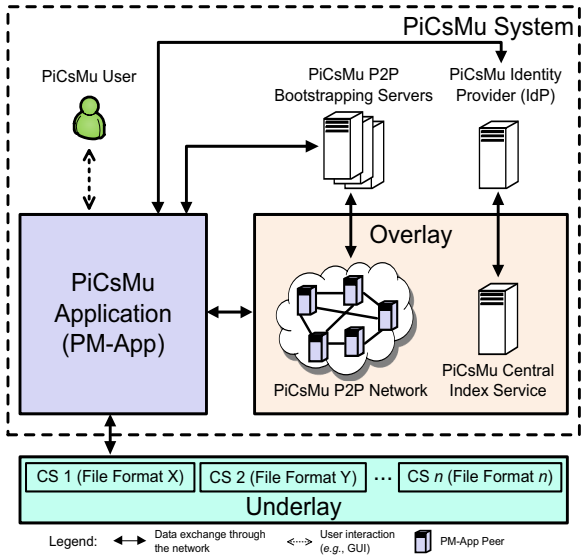

Fig. 1. PiCsMu System Architecture.

A. PiCsMu System Architecture

Fig. B illustrates the PiCsMu System architecture and its interactions with external entities. The PiCsMu System is divided into a *PiCsMu Application*, *PiCsMu P2P Bootstrapping Servers*, a *PiCsMu Identity Provider (IdP)*, and a *PiCsMu overlay*. The PiCsMu Application is responsible for providing to PiCsMu Users means to upload, download, and share files, also, providing an interface to create a PiCsMu Identity. A PiCsMu Identity is created before using the PiCsMu Application. The PiCsMu P2P Bootstrapping Servers provide means to the PiCsMu Application to join the PiCsMu P2P network. Thus, at least one PiCsMu P2P Bootstrapping Server has to be known by the PiCsMu Application. During a file upload and download processes, the application interacts with the PiCsMu Overlay and the underlay.

The PiCsMu Overlay is composed out of the PiCsMu P2P Network and the PiCsMu Central Index Server, both storing file index information. The index keeps track of files stored within PiCsMu, where the file data is located, and how to access it (*cf.* Section. III-B). The PiCsMu P2P Network is maintained to persist index information of shared files. A P2P network was chosen due to scalability and performance metrics, as well as to avoid a single point of failure. The PiCsMu Central Index Server persists index information of private files, *i.e.*, files that are not meant to be shared with other PiCsMu Users, but to be kept for private use. A centralized index server was chosen, since a controlled environment represents a more reliable system (in terms of stability and data availability), if compared to P2P networks. Centralized index servers do not share content to others, avoiding index information being distributed several times to multiple PiCsMu Users, thus, possibly impacting the servers’ performance. The underlay, which is not provided by the PiCsMu System, is composed out of one or multiple CSs, each associated with a valid credential. While the overlay stores references to data parts, the underlay stores the actual data. For downloading files CSs credentials are not necessary, however, to store and/or share files, the PiCsMu User must provide at least one CS’s credential beforehand. In order to use the PiCsMu System a user has to create a PiCsMu Identity within a PiCsMu IdP using the PiCsMu Application. A PiCsMu Identity consists of the PiCsMu identifier (PM-ID), the PiCsMu public key, and an encrypted PiCsMu private key. The PM-ID is a unique identifier (*e.g.*, username) to distinguish PiCsMu Users within the PiCsMu System. The public and private keys are generated, when the PiCsMu User creates his/her identity. For the key generation, the PiCsMu Application uses RSA, with a key length of 1,024 Byte. The application uses Password-based Encryption (PBE) to encrypt the private key and, therefore, persists the encrypted private key within the PiCsMu IdP. The advantage of encrypting the private key with a password is to prevent the user on maintaining a private key locally (most of times persisted in a hard disk), which can represent a security exposure. However, if the PiCsMu User loses such password, another key pair must be generated, and the PiCsMu IdP should be contacted. Moreover, it is also possible to set pre-existing RSA keys to create a PiCsMu Identity.

B. PiCsMu Index Information

The PiCsMu Index consists out of information entities of the PiCsMu file upload process result and contains all parameters necessary to locate and reconstruct a file within the PiCsMu System. The PiCsMu Index consists of three independent top-level entities: the *File Information*, *Credential Information, File Part Information*. Only with all three entities the PiCsMu Application can find all corresponding file parts and can reconstruct the original file.

File Information: The description of a file in the PiCsMu System. Each file is identified by a Universally Unique Identifier (UUID), using UUID version 4 [6], relying on random numbers. In addition, mandatory information such as file name, file size, upload date, and number of file parts are included. Optional information as description and tags may be included by the PiCsMu User.

Credential Information: The CS credential where the encoded file part was stored. The credential information is composed of a unique identifier (generated using a UUID), the credential type ("OAuth" or "username/password"), the CS that it belongs to, the credential itself, which can be an OAuth token or username/password string, and the credential expiration time. Credential Information instances are associated to one or more File Part Information entities through the Credential Information unique identifier.

File Part Information: Those file information required: a unique identifier, a credential identifier, and a file part size. The file part unique identifier is composed out of the file UUID concatenated with the file part order identifier, starting from "0". Therefore, each file part is considered unique by the PiCsMu System relying on the file UUID information plus the file part order (*cf.* Section. IV-C). The credential identifier, which also is a UUID, points to an existing Credential Information entity. The File Part Information instances are associated to one or more File Information instances also through the File Information unique identifier, and, if a File Part Information instance is updated (*e.g.*, the file part was upload to a different CS due to fault tolerance reasons), it does not require to update the File Information instance. Further elements of the File Part Information are: *Encryption Information*, *Encoding/Decoding Information*, and *CS Information*:

Encryption Information: The encrypting information is composed out of a salt, an Initialization Vector (IV), and a password. Each file part is encrypted separately, using a PBE method, with a randomly generated IV and password.

Encoding/Decoding Information: The description of the encoder/decoder used for each encrypted file part. Each encrypted file part is encoded separately, with the encoder being chosen based on which CS the resulting encoded file part will be uploaded to, as well as based on the CS file format restriction. Hence, the used encoding algorithm and the embedded file format have to be placed in the index.

CS Information: Knowledge of where an encoded file part was stored. This includes individual locations (*i.e.*, URL or internal CS's unique identifiers as, *e.g.*, a picture identifier within a Google Picasa account) of all CSs used during the upload process.

C. File Upload and Download Processes

Fig. 2 shows the PiCsMu file upload process. A PiCsMu User selects a file to upload, for sharing or private use. The user is responsible for providing valid CSs credentials. Once the file and CSs credentials are set, the PiCsMu application continues with (1) fragmentation, (2) encryption, (3) data encoding, and (4) data upload to those CSs, which are based on the CSs credentials the user has provided. Within step (1), files are split generating *file parts*. Each of these file parts is encrypted in (2), resulting in *encrypted file parts*. Step (3) encodes each of the encrypted file parts using a data encoder, which converts the file part into a specific file format (each file format chosen based on available CSs credentials and implementation decisions), resulting in an *encrypted and encoded file part*. *E.g.*, data can be encoded into an image file through the means of steganography [9]. Finally, the file part encoded is uploaded to a CS in step (4), using the CSs' API.

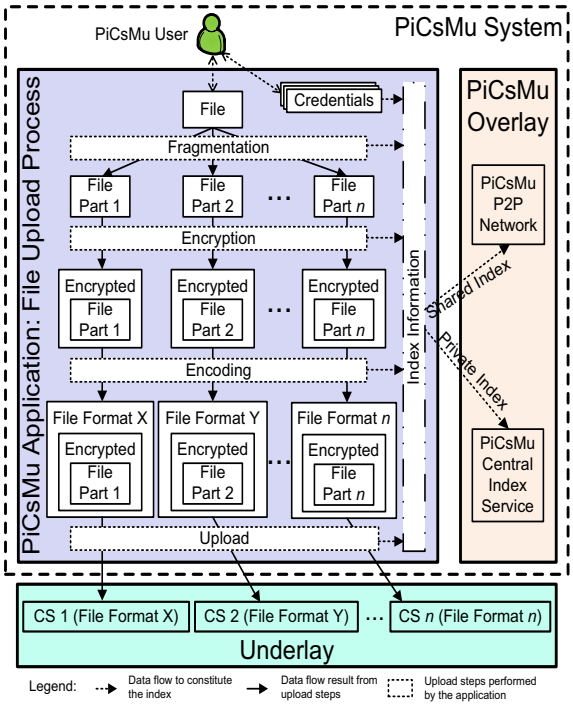

Fig. 2. File Upload Process.

The information produced in steps (1), (2), (3), and (4) forms the PiCsMu index. Once the PiCsMu Application uploads all encoded files to CSs, the user chooses where to persist the generated index. While for private storage the PiCsMu System uses the centralized index server to persist private index information, for shared files the PiCsMu Application uses a DHT. Using the PiCsMu Index information, the PiCsMu System is able to retrieve and reconstruct a previously stored file, applying the reverse process: *downloading* encoded files from one or many CSs, *decoding* these files to obtain encrypted file parts, *decrypting* them to obtain plain file parts data and, finally, *joining* all file parts to acquire the original file that was stored.

D. Private Storage

If a file should not be shared, *i.e.*, kept private to the uploading PiCsMu User (private mode), the PiCsMu Index information of the file is stored in the PiCsMu Central Index Server. Thus, it is responsible for authenticating and authorizing which PiCsMu User has access to which PiCsMu Index entity. The authentication is performed by the PiCsMu IdP. The PiCsMu Central Index Server receives an authentication request from the PiCsMu Application containing the PiCsMu Identity (username and password). Thus, the PiCsMu IdP can check the validity of the given PiCsMu Identity. Once the authentication is done, the PiCsMu Central Index Server can authorize the user to have access to previously persisted private index entities. Therefore, the PiCsMu System guarantees that each PiCsMu User has the right to only delete or retrieve its own file index entities.

E. Private Sharing

Within the private sharing mode, a PiCsMu User (sender) can upload and share a file with one or more specific PiCsMu Users (receivers). Moreover, receivers have the possibility to verify the sender within the PiCsMu IdP. The private sharing mode and the sender verification mechanism are described in 4 steps:

1) Create a Sender's Digital Signature: The sender creates a signature based on the Index entity with the sender's private key.

2) Encrypt the PiCsMu Index entity: The sender encrypts the top-level PiCsMu Index entity using a *randomly generated key* (PBE algorithm).

3) Encrypt the Random Generated Key: The sender encrypt the *randomly generated key* from step 2 with the public keys from the receivers.

4) Concatenate Results: All results of steps 1 to 3 are appended. The result of these steps applied for each PiCsMu Index entities generates PiCsMu Encrypted Index entities. The PiCsMu Encrypted Index entity carries additional information and, therefore, ensures that the receiver obtains all cryptographic information necessary.

Every time a file is shared with another PiCsMu User, the sender is responsible for generating a PiCsMu Share Notification entity for each receiver. The PiCsMu Share Notification entity is also stored in the DHT, using the PM-ID as the DHT key. The information contained in the PiCsMu Share Notification entity is the shared file UUID and the sender PM-ID. PiCsMu uses a friendship relation (unilaterally acknowledged, as publish/subscribe). The friends list, which is a set of friend relations, is kept in the PiCsMu IdP, since it could be manipulated by malicious peers if persisted in the PiCsMu P2P Network. The friends list is fetched from the PiCsMu IdP and transmitted to the PiCsMu Application upon start and the list is updated on the PiCsMu IdP, if a modification happens on the application side (*e.g.*, add/remove a friend). The friends list, which is composed out of other PiCsMu Users, is mandatory, if the PiCsMu User aims to receive shared files. To avoid fake share notifications, a PiCsMu User only can receive files, if the sender's PiCsMu Identity exists in his/her friend list. If the PiCsMu System is only used for private storage or to share files with others (but not receive shared files), the friends list is optional.

F. Public Sharing

Within the public sharing mode, a PiCsMu User can upload and share a file with the entire PiCsMu System and, therefore, any PiCsMu User has access to it. In this mode no index encryption is needed. Thus, each of the PiCsMu Index entities are directly stored in the DHT.

Instead of using PiCsMu Share Notifications, content can be searched through a *content-based search* query. Content-based search used in PiCsMu allows for the lookup of files without knowing an exact keyword by using P2PFastSS [3], a similarity algorithm based on the edit distance metric (Levenshtein) [10]. The search is enabled by storing multiple keywords as DHT keys, pointing to the corresponding PiCsMu Index entities. These keywords are generated based on the description and tags attributes of the File Information entity. Multiple keywords are stored for each uploaded file. Therefore, the search is performed by finding keywords entered by the PiCsMu User that are in the DHT as keys.

G. Use Case

To demonstrate the file upload and download processes, a use case is provided for the private file sharing. Fig. 3 illustrates Alice sharing a file specifically with Bob. "PM-ID" represents the unique identifier of a PiCsMu Identity, having "PM-ID A" as Alice's and "PM-ID B" as Bob's identifiers.

Both Bob and Alice start their PiCsMu Application by fetching her and his public and encrypted private key, respectively, from the PiCsMu IdP (step 1). The encrypted private key is decrypted with that user password, which was chosen when the PiCsMu Identity was created to obtain the private key. In parallel, also in step 1, their respective PiCsMu Applications contact the PiCsMu P2P Bootstrapping Servers to connect to the PiCsMu P2P Network.

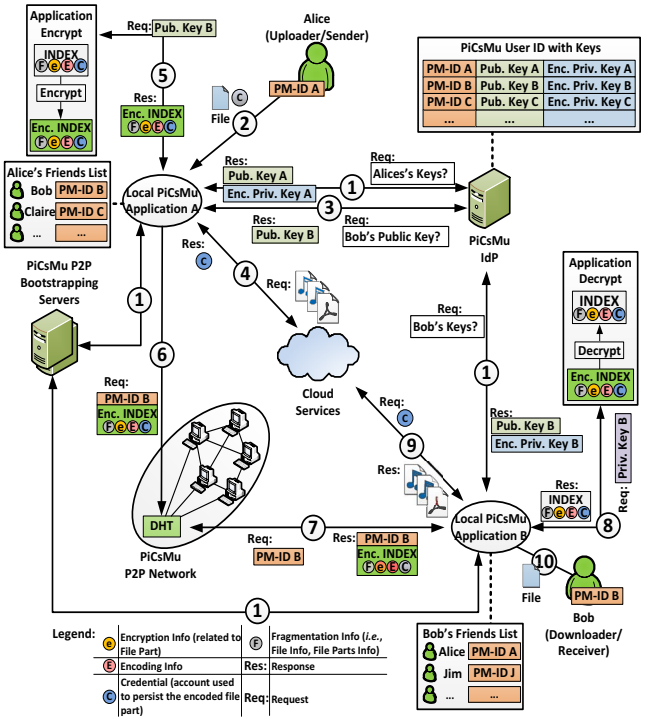

Fig. 3. Private sharing steps.

In step 2 Alice provides to her PiCsMu Application the file to be shared with Bob and one or more CSs' credentials. Once the PiCsMu Application receives the file and all CSs credentials, it starts the file upload process as described in Section. III-C. During the file upload process her PiCsMu Application recognizes that the file is shared (with Bob) and within step 3 Alice's PiCsMu Application requests Bob's public key from the PiCsMu IdP, including Bob's PM-ID.

At step 4 Alice's PiCsMu Application uploads the encoded file parts to those CSs Alice provided credentials for.

Once the uploading of all file parts is successfully completed, Alice's PiCsMu Application encrypts each PiCsMu Index entities with a randomly generated key, which is also encrypted with Bob's public key (step 5). Moreover, Alice's digital signature is included with each of the PiCsMu Encrypted Index entities.

In step 6 Alice's PiCsMu Application persists each of the PiCsMu Encrypted Index entities to the DHT. Therefore,

Alice creates a PiCsMu Share Notification entity to make Bob aware of the newly shared file.

Bob's PiCsMu Application consults the DHT for PiCsMu Share Notification entities that was sent by his friends. Once Bob recognizes a new notification, here from Alice, Bob's PiCsMu Application uses the file UUID from the PiCsMu Share Notification to start a sequence of DHT calls that are necessary to retrieve all PiCsMu Encrypted Index entities that forms the PiCsMu Index (step 7).

Holding all PiCsMu Encrypted Index entities of the shared file, Bob's PiCsMu Application verifies in step 8 the digital signature from step 5 against Alice's public key provided by the PiCsMu IdP. In parallel, Bob's PiCsMu Application is able to decrypt the PiCsMu Encrypted Index entities by using Bob's private key.

Finally, within step 9 all CSs' credentials, part of the index, are used to locate all encoded file parts in the CSs storage. Upon retrieving all encoded file parts, Bob's PiCsMu Application follows the file download process and reconstructs the original file shared by Alice.

IV. PiCsMu Prototype

A Java-based prototypical implementation of the PiCsMu System was developed to verify its feasibility, scalability, and overhead. The PiCsMu Application implements the following modules using well defined interfaces for reusability: *Schedulers*, *DataEncoders*, *EncryptionStrategies*, *IndexPersistenceStrategies*, and *CloudServices*. The security modules implement *PBE-AESCryptography*, which is used to encrypt larger amounts of data, such as PiCsMu Index entities, while the *RSACryptography* just encrypts smaller amounts of data, such as a randomly generated key. The *PBE-AESCryptography* handles a key length up to 128 bit.

CentralizedIndex and *DistributedIndex* implement the persistence of PiCsMu Index entities in the PiCsMu Central Index Server and in the PiCsMu P2P Network, respectively. Both implement methods to persist PiCsMu Index entities, such as *putFileInfo()*, *putFilePartInfo()*, *putCredentialInfo()*, as well as methods to retrieve the PiCsMu Index entities, such as *getFileInfo()*, *getFilePartInfo()*, and *getCredentialInfo()*. The *CentralizedIndex* implementation expects as input the PM-ID and PiCsMu Identity password for each *get* or *put* called methods, since the PiCsMu Central Index Server is implemented with stateless calls (*i.e.*, without the notion of a session). Therefore, *get* or *put* methods can be invoked any time in a Remote Procedure Call (RPC) fashion encoded in JSON (JavaScript Object Notation). The communication between the *CentralizedIndex* implementation and the PiCsMu Central Index Server is done following the JSON-RPC standard [8]. The PiCsMu Central Index Server and the PiCsMu IdP are implemented as standalone Java programs, receiving and answering JSON-RPC method calls. In the current prototype implementation, both PiCsMu Central Index Server and PiCsMu IdP run a MySQL database to persist all PiCsMu Index entities and PiCsMu Identities. The *DistributedIndex* implementation uses the TomP2P library [20] allowing the PiCsMu Application to put or get PiCsMu Index entities, search keywords, and PiCsMu Share Notification entities to/from the DHT. TomP2P has the notion of domains, which are used together with a DHT key to build a unique combination. Therefore, the *DistributedIndex* implementation uses three DHT domains: (1) index domain, (2) search domain, and (3) share notification domain. In (1) only PiCsMu Index entities are stored; (2) is used to store search keywords pointing to PiCsMu Index entities; and (3) stores PiCsMu Share Notification entities. The PiCsMu Bootstrapping Servers are also implemented as standalone Java programs to receive bootstrapping requests from the PiCsMu Application.

A. Data Encoders

A File Format Sample (FFS) is a file with a specific format that is used to encode data into. *E.g*, in order to inject data into MP3 files, a MP3 FFS is necessary, as well as a *data encoder*, which is the software that performs the *encoding* using the FFS.

Each of the data encoders supported by PiCsMu were implemented having three basic methods as the interface to the PiCsMu Application: *pack()*, *unpack()*, and *getEstimatedOutputFileSize()*. The *pack()* method expects the data to be encoded as input, performs the encoding process, and provides the encoded file as output. The *unpack()* does the decoding process. The *getEstimatedOutputFileSize()* method performs an estimation of the final encoded file size, without actually encoding any data in a FFS. As parameter, the method expects a data size as input to the encoder in order to perform the estimation.

The following data encoders are supported by the PiCsMu system prototype:

IDv3 Tag Encoder: This encoder injects data using optional headers of an audio file format. The ID3 Tag Encoder uses the ID3 version 2 [7] metadata container to inject data in all fields specified by the standard, which are, *e.g.*, title, artist, album, track number, among others. The standard specifies a metadata tag size up to 256 MByte. Thus, this encoder uses a MP3 File Format Sample up to 10 seconds of audio and encodes data in the ID3v2 tag within the file, where the encoded data can vary depending on the amount of data that the *Scheduler*. The output is a playable MP3 file with the specified data in ID3v2 tags.

Text Encoder: This encoder uses a data conversion technique to transform data into a set of ASCII characters. This encoder is able to upload the encoded ASCII data to CSs that only accept textual input. It takes an amount of data as input, and generates as output a sequence of ASCII characters. Therefore, the Text Encoder does not require a FFS.

JPG and PNG Steganography Encoder: This encoder uses the steganography technique, where data is hidden in a way that intends to turn them imperceptible apart from the sender and receiver. The purpose of this encoder is to inject data (hiding it) in JPG and PNG files. It takes a FFS (JPG or PNG) as input and, for each pixel, it injects 3 bits of data (also specified as input) into the LSBs (Least Significant Bit). The output is a JPG or PNG FFS, which includes the data specified. There are more sophisticated image steganography methods already widely discussed [9].

B. Cloud Services

Each of the CSs supported by PiCsMu were implemented having 4 basic methods as the interface to the PiCsMu Application: (a) *put()*, (b) *get()*, (c) *getMaxInputSize()*, and (d) *getAllowedFileFormats()*. In (a) and (b), the CS implementation should be able to store and retrieve files, respectively, having a CS credential as parameter. While in (a) the method returns a unique identifier to locate the file within the scope of the CS account associated with the specified CS credential, in

(b) the method expects this unique identified as parameter to retrieve the correct file. In (c) the implementation specifies what is the maximum file size that this CS can handle for each upload, and in (d) the CS implementation returns the allowed file formats to be uploaded. Based on the allowed file formats the PiCsMu Application can check what are the possible data encoders to be used with each CS. The following CSs are supported by the PiCsMu system prototype:

CSG Service: The CSG Service is an internal storage service in the CSG group at the University of Zurich. The *CSGService* implementation uploads a file, without file format restrictions, up to 20 MByte, and it returns a URL with a randomly generated identifier to fetch the file again, without the need of a CS credential. Credentials are necessary to upload, but not to download files. The CSG Service is classified as a generic Cloud storage service.

Google GMail: Google GMail is an email service which allows to store data within the email body and email attachments. *GMailService* implementation uses the structure of an email to store data, thus being classified as a data-specific Cloud storage service. It creates an email to be sent to the same GMail account associated with the specified CS credential, storing the encoded file as an attachment. The attachment maximum file size is 25 MByte, and the email body is left blank. In order to identify in which email the encoded file is located, the *GMailService* implementation generates a random UUID in the email subject. The *GMailService* implementation accepts any format.

Facebook Update Status: Facebook is a social network service providing the status update as a feature. Facebook users can write a text input which has a maximum input size of 63,206 characters by today. The *FacebookStatusUpdateService* implementation uses the status update field to store data, thus, being classified as a data-specific Cloud storage service. Every time that a status update is created, the CS generates a unique identifier associated with the status updated. Therefore, data represented as text is the input, while a URL generated by the CS that locates the status updated is the output. The CS credential used to update the status is also necessary to retrieve the status update. *FacebookStatusUpdate Service* only accepts text as input.

Google Picasa: Google Picasa is an image organizer and viewer. Google Picasa accepts the upload of images with sizes up to 20 MByte, despite of the image resolution, thus, being classified as data-specific Cloud storage service. The *PicasaService* implementation takes an image file as input, generates a random UUID which is associated with the Picasa image name attribute, and returns such a UUID as an output. The UUID is valid in the scope of the CS credential given to upload the image, therefore, the used CS credential to upload is also necessary to download the file. Although the Google Picasa service accepts a wide range of image file formats, the *PicasaService* implementation prototype at this stages accepts PNG and JPG formats only.

ImageShack: ImageShack is an image organizer and viewer. It accepts the upload of images with sizes up to 5 MByte, despite of the image resolution, thus, being classified as data-specific Cloud storage service. The *ImageShackService* implementation takes an image file as input and returns a URL with a randomly generated identifier to fetch the file again, without the need of a CS credential. A CS credential is necessary to upload, but not to download the file. The *ImageShackService* implementation today accepts PNG and JPG files, even though the CS accepts a wider variety of image formats.

SoundCloud: SoundCloud is a service platform which allows users to upload, organize, and listen to audio. It accepts the upload of audio files, without a maximum file size, but with a free account limiting to 60 minutes of audio. SoundCloud is classified as being a data-specific Cloud storage service. The SoundCloudService implementation takes an audio file as an input and returns a URL with a randomly generated identifier to fetch the file again. The CS credential used to upload is also necessary to download the file. Currently, the *SoundCloudService* implementation accepts MP3 files today, even though the CS accepts generally a wider variety of audio formats.

C. Scheduler Algorithm

The PiCsMu Scheduler module chooses how to fragment files, which encryption to use, which data encoder to select, and finally to which CS to upload. Different Scheduler implementations may prioritize different aspects, *e.g.*, to lowest number of file parts, to lowest number of CS, or to highest number of CS. The *DefaultScheduler* implementation uses a random function to determine CSs and its credentials, data encoders, and encryption strategies for each file part. The fragmentation strategy followed by the *DefaultScheduler* is to try to put as much data within one file part, considering the chosen CS and data encoder. The *DefaultScheduler* consults the chosen data encoder again to have an estimate of what will be the final file size after the encoding process (using *getEstimatedOutputFileSize()*). Based on the estimated file size after the encoding process, the *DefaultScheduler* checks if the estimated file size can be uploaded to the chosen CS (using the CS implementation *getMaxInputSize()* method). If not, the *DefaultScheduler* has to decrease the file part size, to be given as input to the data encoder. Since the estimated output size is often correct and potential errors are small, the current implementation decreases the file part size by 10 Byte, in a loop, before the estimation is made again and checked if the newly estimated file size can be uploaded to the chosen CS.

V. Evaluations

The evaluation was performed in a controlled environment, with 17 physical nodes interconnected to one isolated Gigabit switch. The node *n1* holds both the PiCsMu Central Index Server and the PiCsMu IdP. The PiCsMu P2P Bootstrapping Server is hosted in the node *n2*. All PiCsMu Application instances are pointed to *n1* to store private indices and retrieve PiCsMu Identities, and to *n2* to discover other peers in the PiCsMu P2P Network. In addition, the node *n3* runs the CSG Service, which emulates CSs. Nodes *n1*, *n2*, and *n3* have a Intel Xeon processor, with 2.2 GHz, 48 GByte RAM, and 500 GByte 7200 RPM hard drive. Although PiCsMu has been tested and its functionality has been verified using public CSs, the decision to not run evaluations with all supported CSs' implementation relies on two reasons: first, performing evaluations with a local service within a local network isolate the results, thus excluding, *e.g.*, delays in public networks that could affect overall measurement values; second, the evaluation with a local service enables more test runs due to be performed in a controlled environment. The remaining 14 nodes (AMD Opteron, 24 cores, 2.5 GHz, with 64 GByte RAM, and 500 GByte 7200 RPM hard drive) were used to run PiCsMu Application instances, which required about 40 MByte of

RAM allowing a maximum of 700 active PiCsMu Application instances (*i.e.*, different PiCsMu Users). All PiCsMu Application instances are connected through a local network and listen on different ports and all 700 PiCsMu Application instances are always online, thus, no churn is considered. In all experiments all PiCsMu Application instances use the same CS service but with different CS credentials. The same files are used for each test case and test cases are repeated 10 times.

The following dimensions are observed: total time for file upload/download (including all file upload and download steps within the PiCsMu Application), file data overhead percentage (how many percent the original file grows, looking to the encoded file parts size as a result of the encoding), and PiCsMu Index overhead (how much data was generated by the PiCsMu Index entities). These dimensions determine the feasibility, scalability, and overhead of the PiCsMu System.

TABLE III: TEST CASES AND SCENARIOS.

Scenarios	Test case A	Test case B
(1) Private Storage mode	1 PiCsMu User uploading a private file	1 PiCsMu User download a private file
(2) Private Sharing mode	1 PiCsMu User uploading a file to share specifically with all other PiCsMu Users part of the PiCsMu IdP	All PiCsMu Users downloading the shared file
(3) Public Sharing mode	1 PiCsMu User uploading a file to share to the whole PiCsMu P2P Network	All PiCsMu Users part of the P2P Network downloading a file that was shared to the whole PiCsMu P2P Network
(4) CSGService Without PiCsMu	1 User uploading a file to the CSG Service	1 User downloading a file from the CSG Service

A. Evaluation Test Cases

The following test cases (*cf.* Table III) were designed to cover the three modes of operation related to the file upload and download processes: (1) private storage, (2) private sharing, and (3) public sharing. Moreover, an additional test case was compiled to generate results for a comparative analysis: (4) CSGService without PiCsMu System. Test case (4) shows results of the file upload and download without the use of the PiCsMu System, which means that no additional overhead or processing time is produced. These test cases used files with sizes of 1 MByte, 10 MByte, 100 MByte, and 1 GByte in order to upload and download.

The total time to upload and download are measured in test cases (A) only, since it generates the same (for scenario 1 and 4) or higher overhead/total time (for scenarios 2 and 3) as (B), representing the upper bound. Scenario 2 issues share notifications for all its users, resulting in a higher overhead than scenario 3, also representing the upper bound.

B. Evaluation Results

Fig. 4 shows the total time for scenarios (1), (2), and (4). In scenario (1) a 1 GByte file can be uploaded on average in 11.1 minutes (test case A) and downloaded in 6.7 minutes (test case B). When compared to the average total time of scenario (4), with the same 1 GByte file, where the upload is completed in 1.59 minutes (test case A) and download is completed in 1.53 minutes (test case B), a PiCsMu User takes on average 6.9 times to upload and 4.3 times to download. This is due to fragmentation, encoding, and encryption overhead. In scenario (2), the upload and download process is faster, since the index is not encrypted and no DHT calls are performed. Another observation is that with an increasing number of file parts, the total time is increasing too, since the index for each file part is encrypted individually.

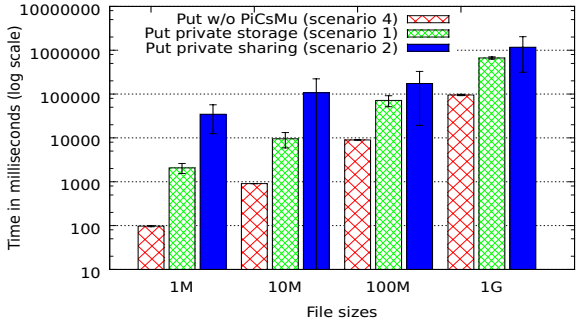

Fig. 4. Total Times for Test Case A and Scenarios 1, 2, and 4.

Fig. 5 shows the data overhead for test case A of scenarios (1) and (2) with a considerable data overhead variance for smaller files when compared to the 1 GByte file. Since the PiCsMu Application uses a random function, there is a higher chance to choose different data encoders for less fragments in different runs. Data encoders present different overhead. The best encoder (*SteganographyEncoder*) has an overhead of 0.16% due to compressing the FFS (PNG image) after the 3 LSBs are injected, while the worst encoder (*ID3v2Tag Encoder*) has an overhead of around 104%. The *ID3v2Tag Encoder* is the one with most overhead since data is injected and represented in a hexadecimal String format within ID3v2 tags, *e.g.*, album name, song title, song description, etc. However it is also the encoder that stores the most data (5 MByte, due to CSG Service upload size restrictions, compared to 43 KByte with *SteganographyEncoder*). For large files, the overhead levels off at around 100%, which is the average over all encoders.

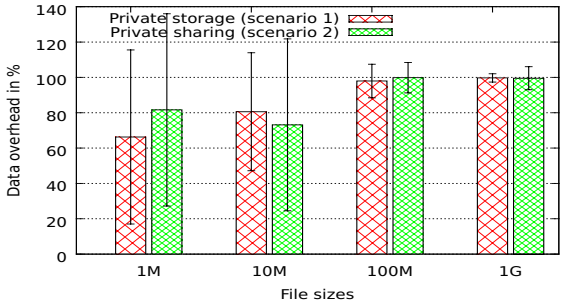

Fig. 5. File Data Overhead.

Two distinct runs with extreme variations in the number of files parts were chosen to analyze the relation between data encoders and data overhead. Considering the 1 GByte file, the PiCsMu Application generated 572 encoded file parts, with a total size of 1.97 GByte data being uploaded. In a second run it resulted in 237 encoded file parts, with a total size of 2.03 GByte, resulting in an additional overhead of 56 MBytes.

Fig. 6 shows the index overhead for scenario 2 and 3. The index size for (2) is much smaller than for the private sharing, since the index does not need to be distributed in the DHT. For (3) the overhead to store the index in the DHT is

between a factor of 2.4 to 2.7 explained by DHT's redundancy settings. It can be observed that the index overhead levels off at around 2,400 Byte per peer for the private sharing, respectively 900 Byte for the private storage.

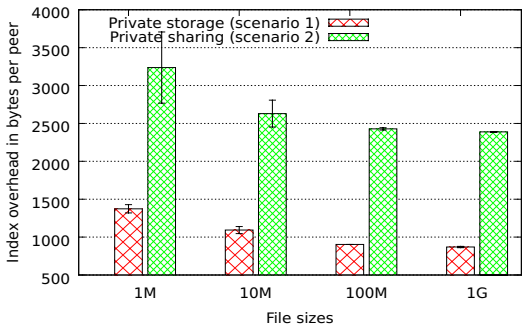

Fig. 6. PiCsMu Index Overhead.

VI. Summary and Future Work

PiCsMu defines a novel storage overlay aggregating heterogeneous Cloud storage services, supporting both generic and data-specific storage service types. For data-specific Cloud storage services the lack of data validation is exploited to provide a generic storage service. PiCsMu provides a single interface to end users enabling storage and data sharing.

PiCsMu shows advantages in terms of increased security and privacy: the data encoding, file part encryption, and file fragmentation processes, combined, add another layer of tasks to reconstruct original files, thus, turning it even harder for an attacker to gain access to the content of original files. Another advantage is data redundancy: multiple fragments in multiple CSs prevents data loss in case that a single CS provider shuts down its services like, *e.g.*, Megaupload [21]. Furthermore, additional storage space is available with PiCsMu, since multiple CS that offer free storage can be aggregated.

The evaluation addressed all research questions proposed in Section. I. The PiCsMu prototype implementation showed that it is possible to build an overlay over generic and data-specific CS storage, used to store, retrieve, and share any kind of files. The total time for storing and retrieving data and its overhead showed that PiCsMu scales with respect to different file sizes. The overhead with respect to the index is moderate even for 700 users, all receiving a share notification.

These experiments indicated that the Scheduler is a key component as it decides how many fragments are generated. Prioritizing to generate less file parts results in less overhead, but it does not spread the data to many CSs, especially not for small files. Thus, for future work, the Scheduler should take the file size into consideration. Furthermore, the Scheduler could decide which CS to take, based on the current performance of the CS in order to optimize PiCsMu for the end user. Since this optimization considers the storage of data, the time of retrieval of the data is not known in advance and could be anticipated based on previous user behavior.

This paper does not consider any legal implications, such as boundaries of Cloud storage services. Thus, the PiCsMu User has the responsibility to decide about legal aspects, whether the use of a Cloud storage overlay system is conforming (*i.e.* fair-use) to the terms of service and its legal intent of CS providers. Additionally, such legal aspects will be explored in the future, including views of various stakeholders and their interests. Moreover, additional evaluation measurements showing the system's effectiveness will be carried out, using the public CSs supported by PiCsMu.

VII. Acknowledgements

This work was supported partially by the SmartenIT and the FLAMINGO projects, funded by the EU FP7 Program under Contract No. FP7-2012-ICT-317846 and No. FP7-2012-ICT-318488, respectively.

References

[1] Amazon.com Web Services: *Products and Services*. Available at: http://aws.amazon.com/products. Last visited on: March 2013.

[2] M. Armbrust, A. Fox, R. Griffith, A. D. Joseph, R. H. Katz, A. Konwinski, G. Lee, D. A. Patterson, A. Rabkin, I. Stoica, M. Zaharia: *Above the Clouds: A Berkeley View of Cloud Computing*. Technical Report UCB/EECS-2009-28, University of California at Berkeley, California, U.S.A, February 2009.

[3] T. Bocek, E. Hunt, D. Hausheer, B. Stiller: *Fast Similarity Search in Peer-to-Peer Networks*. 11th IEEE/IFIP Network Operations and Management Symposium (NOMS 2008), Salvador, Brazil, April 2008, pp. 240-247, doi: 10.1109/NOMS.2008.4575140.

[4] Dropbox: *Dropbox Service*. Available at: dropbox.com. March 2013.

[5] Google Corporate: *Google Products and Solutions*. Available at: http://google.com/intl/en/about/products/. Last visited at: March 2013.

[6] IETF: *A Universally Unique IDentifier (UUID) URN Namespace*. RFC 4122, July 2005. http://tools.ietf.org/html/rfc4122, March 2013.

[7] ID3 Website: *ID3 Version 2 Standard*. Available at: id3.org. May 2013.

[8] JSON-RPC Website Specification: *A Light Weight Remote Procedure Call Protocol, Specification 2.0*. Available at: jsonrpc.org. April 2013.

[9] G. C. Kessler: *An Overview of Steganography for the Computer Forensics Examiner*. Technical Report, Gary Kessler Associates, February 2004. http://www.garykessler.net/library/fsc_stego.html.

[10] V. I. Levenshtein: *Binary Codes Capable Of Correcting Deletions, Insertions And Reversals*. Soviet Physics Doklady, Vol. 10, No. 8, February 1966, pp. 707-710.

[11] Z. Li, G. Xie, K. Hwang, Zhongcheng Li: *Churn-Resilient Protocol for Massive Data Dissemination in P2P Networks*. IEEE Transactions of Parallel and Distributed Systems, Vol. 22, No. 8, August 2011, pp. 1342-1349. doi: 10.1109/TPDS.2011.15.

[12] E. K. Lua, J. Crowcroft, M. Pias, R. Sharma, S. Lim: *A Survey and Comparison of Peer-to-Peer Overlay Network Schemes*. IEEE Communications Surveys and Tutorials, Vol. 7, No. 2, pp. 72-93, 2005.

[13] G. S. Machado, F. Hecht, M. Waldburger, B. Stiller: *Bypassing Cloud Providers Data Validation To Store Arbitrary Data*. 13th IFIP/IEEE International Symposium on Integrated Network Management (IM 2013), Ghent, Belgium, May 2013.

[14] Microsoft: *SkyDrive, Files Anywhere*. http://skydrive.com. March 2013.

[15] OAuth Specification: *OAuth 2.0 Specification*. Available at: http://oauth.net/2. Last visited on: March 2013.

[16] Otixo: *All Your Cloud Files from a Single Login*. Available at: http://otixo.com. Last visited on: March 2013.

[17] PiCsMu Website: *Platform-independent Cloud Storage System for Multiple Usage*. Available at: http://www.pics.mu , May 2013.

[18] SalesForce.com: *The Leader of Customer Relationship Management (CRM) and Cloud Computing*. Available at: http://www.salesforce.com. Last visited on February 2010.

[19] SpiderOak: *Private Online Backup and Sharing*. Available at: https://spideroak.com. Last visited on: March 2013.

[20] TomP2P Website: *A P2P-based High Performance Key-value Pair Storage Library*. Available at: http://tomp2p.net, March 2013.

[21] Wikipedia: *Megaupload*. Available at: http://en.wikipedia.org/wiki/Megaupload. Last visited on: March 2013.

[22] Wuala: *Secure Storage Service*. Available at: wuala.com. March 2013.