DepSky: Dependable and Secure Storage in a Cloud-of-Clouds

Alysson Bessani, University of Lisbon, Faculty of Sciences
Miguel Correia, Instituto Superior Técnico / INESC-ID
Bruno Quaresma, University of Lisbon, Faculty of Sciences
Fernando André, University of Lisbon, Faculty of Sciences
Paulo Sousa, Maxdata Informática

The increasing popularity of cloud storage services has lead companies that handle critical data to think about using these services for their storage needs. Medical record databases, large biomedical datasets, historical information about power systems and financial data are some examples of critical data that could be moved to the cloud. However, the reliability and security of data stored in the cloud still remain major concerns. In this work we present DepSky, a system that improves the availability, integrity and confidentiality of information stored in the cloud through the encryption, encoding and replication of the data on diverse clouds that form a cloud-of-clouds. We deployed our system using four commercial clouds and used PlanetLab to run clients accessing the service from different countries. We observed that our protocols improved the perceived availability and, in most cases, the access latency when compared with cloud providers individually. Moreover, the monetary costs of using DepSky on this scenario is at most twice the cost of using a single cloud, which is optimal and seems to be a reasonable cost, given the benefits.

Categories and Subject Descriptors: D.4.5 [**Operating Systems**]: Reliability–Fault-tolerance; C.2.0 [**Computer-Communication Networks**]: General–Security and protection; C.2.4 [**Distributed Systems**]: Distributed applications

General Terms: Algorithms, Measurement, Performance, Reliability, Security

Additional Key Words and Phrases: Cloud computing, Cloud storage, Byzantine quorum systems

ACM Reference Format:
Bessani, A., Correia, M., Quaresma, B., André, F., Sousa, P. 2011. DepSky: Dependable and Secure Storage in a Cloud-of-Clouds. ACM Trans. Storage 00, 00, Article 00 (March 2013), 30 pages.
DOI = 10.1145/0000000.0000000 http://doi.acm.org/10.1145/0000000.0000000

1. INTRODUCTION

The increasing maturity of cloud computing technology is leading many organizations to migrate their IT infrastructure and/or adapt their IT solutions to operate completely or partially in the cloud. Even governments and companies that maintain critical infrastructures (e.g., healthcare, telcos) are adopting cloud computing as a way of reducing costs [Greer 2010]. Nevertheless, cloud computing has limitations related to security and privacy, which should be accounted for, especially in the context of critical applications.

This paper presents DepSky, a dependable and secure storage system that leverages the benefits of cloud computing by using a combination of diverse commercial clouds to build a *cloud-of-clouds*. In other words, DepSky is a *virtual storage cloud*, which is accessed by its users to manage *updatable data items* by invoking equivalent operations in a group of individual clouds. More specifically,

A preliminary version of this paper appeared on the *Proceedings of the 6th ACM SIGOPS/EuroSys European Conference on Computer Systems - EuroSys'11*. This work was partially supported by the EC FP7 through project TCLOUDS (ICT-257243), by the FCT through project RC-Clouds (PTDC/EIA-EIA/115211/2009), the Multi-annual Program (LASIGE), and contract PEst-OE/EEI/LA0021/2011 (INESC-ID). This work was done when all authors were at University of Lisbon, Faculty of Sciences. Author's addresses: A. Bessani, B. Quaresma and F. André, Faculdade de Ciências, Universidade de Lisboa, Portugal; M. Correia, INESC-ID, Portugal; P. Sousa, Maxdata Informática, Portugal.

DOI 10.1145/0000000.0000000 http://doi.acm.org/10.1145/0000000.0000000

DEPSKY addresses four important limitations of cloud computing for data storage in the following way:

- **Loss of availability**: When data is moved from the company's network to an external datacenter, it is inevitable that service availability is affected by problems in the Internet. Unavailability can also be caused by cloud outages, from which there are many reports [Raphael 2011], or by denial-of-service attacks like the one that allegedly affected a service hosted in Amazon EC2 in 2009 [Metz 2009]. DEPSKY deals with this problem exploiting replication and diversity by storing the data on several clouds, thus allowing access to the data as long as a subset of them is reachable.
- **Loss and corruption of data**: there are several cases of cloud services losing or corrupting customer data. For example, in October 2009 a subsidiary of Microsoft, Danger Inc., lost the contacts, notes, photos, etc. of a large number of users of the Sidekick service [Sarno 2009]. The data was recovered several days later, but the users of Ma.gnolia were not so lucky in February of the same year, when the company lost half a terabyte of data that it never managed to recover [Naone 2009]. DEPSKY deals with this problem using Byzantine fault-tolerant replication to store data on several cloud services, allowing data to be retrieved correctly even if some of the clouds corrupt or lose it.
- **Loss of privacy**: the cloud provider has access to both the stored data and how it is accessed. The provider may be trustworthy, but malicious insiders are a wide-spread security problem [Hanley et al. 2011]. This is an especial concern in applications that involve keeping private data like health records. An obvious solution is the customer encrypting the data before storing it, but if the data is accessed by distributed applications this involves running protocols for key distribution (processes in different machines need access to the cryptographic keys). DEPSKY employs a secret sharing scheme and erasure codes to avoid storing clear data in the clouds and to improve the storage efficiency, amortizing the replication factor on the cost of the solution.
- **Vendor lock-in**: When a customer hosts its data or services in a cloud, it has to comply with the cloud service provider's billing policy, service characteristics and APIs. In this scenario, there is always the concern that the coupling between the customer and the provider become so high to a point that it becomes economically inviable to move from one provider to another, the so called vendor lock-in problem [Abu-Libdeh et al. 2010]. This concern is specially prevalent in Europe, as the most conspicuous providers are not in the region. Even moving from one provider to another one may be expensive because the cost of cloud usage has a component proportional to the amount of data that is read and written. DEPSKY addresses this issue in two ways. First, it does not depend on a single cloud provider, but on a few, so data access can be balanced among the providers considering their practices (e.g., what they charge). Second, DEPSKY uses erasure codes to store only a fraction (typically half) of the total amount of data in each cloud. In case the need of exchanging one provider by another arises, the cost of migrating the data will be at most a fraction of what it would be otherwise.

The way in which DEPSKY solves these limitations does not come for free. At first sight, using, say, four clouds instead of one involves costs roughly four times higher. One of the key objectives of DEPSKY is to reduce this cost, which in fact it does to about 1.2 to 2 times the cost of using a single cloud. This seems to be a reasonable cost, given the benefits.

The key insight of the paper is that the limitations of individual clouds can be overcome by using a *cloud-of-clouds* in which the operations (read, write, etc.) are implemented using a set of *Byzantine quorum systems protocols*. The protocols require *diversity* of location, administration, design and implementation, which in this case comes directly from the use of different commercial clouds [Vukolic 2010]. There are protocols of this kind in the literature, but they either require that the servers execute some protocol-specific code [Cachin and Tessaro 2006; Goodson et al. 2004; Malkhi and Reiter 1998a; Malkhi and Reiter 1998b; Martin et al. 2002], not possible in storage clouds, or are sensible to contention (e.g., [Abraham et al. 2006]), which makes them difficult to use for geographically dispersed systems with high and variable access latencies. DEPSKY overcomes these limitations by not requiring specific code execution in the servers (i.e., storage clouds), but still being efficient by requiring only two communication round-trips for each operation. Furthermore, it leverages the above mentioned mechanisms to deal with data confidentiality and reduce the amount of data stored in each cloud.

Although DEPSKY is designed for data replication on cloud storage systems, the weak assumptions required by its protocols make it usable to replicate data on arbitrary storage systems such as FTP servers and key-value databases. This extended applicability is only possible because, as already mentioned, the DEPSKY protocols have no server-side specific code to be executed, requiring only basic storage operations to write, read and list objects.

In summary, the main contributions of the paper are:

(1) The DEPSKY system, a storage cloud-of-clouds that overcomes the limitations of individual clouds by using a set of efficient Byzantine quorum system protocols, cryptography, secret sharing, erasure codes and the diversity that comes from using several clouds. The DEPSKY protocols require at most two communication round-trips for each operation and store only approximately half of the data in each cloud for the typical case.

(2) The notion of consistency proportional storage, in which the replicated storage system provides the same consistency semantics as its base objects (i.e., the nodes where the data is stored). DEPSKY satisfies this property for a large spectrum of consistency models, encompassing most of the semantics provided by storage clouds and popular storage systems.

(3) A set of experiments showing the costs and benefits (both monetary and in terms of performance) of storing updatable data blocks in more than one cloud. The experiments were made during one month, using four commercial cloud storage services (Amazon S3, Windows Azure Blob Service, Nirvanix CDN and Rackspace Files) and PlanetLab to run clients that access the service from several places worldwide.

The paper is organized as follows. Section 2 describes some applications that can make use of DEPSKY. Section 3 presents the core protocols employed in our system and Section 4 presents additional protocols for locking and management operations. Sections 5 and 6 show how storage clouds access control can be employed to setup a DEPSKY cloud-of-clouds storage and how the system works with weakly consistent clouds, respectively. The description of the DEPSKY implementation and its experimental evaluation are presented in Sections 7 and 8. Finally, Section 9 discusses related work and Section 10 concludes the paper. The paper also contains a series of appendixes describing some auxiliary functions used in our algorithms (Appendix A), the correctness proofs for the storage (Appendix B) and locking protocols (Appendix C), and a proof of the DEPSKY consistency proportionality (Appendix D).

2. CLOUD STORAGE APPLICATIONS

In this section we briefly discuss some examples of applications that can benefit from a cloud-of-clouds storage system like DEPSKY.

Critical data storage. Given the overall advantages of using clouds for running large scale systems, many governments around the globe are considering the use of this model. The US government already announced its interest in moving some of its computational infrastructure to the cloud and started some efforts in understanding the risks involved in doing these changes [Greer 2010]. The same kind of concerns are also being discussed in Europe [Dekker 2012].

In the same line of these efforts, there are many critical applications managed by companies that have no interest in maintaining a computational infrastructure (i.e., a datacenter). For these companies, the cloud computing pay-per-use model is specially appealing. An example would be power system operators. Considering only the case of storage, power systems have operational historian databases that store events collected from the power grid and other subsystems. In such a system, the data should be always available for queries (although the workload is mostly write-dominated) and access control is mandatory.

Another critical application that could benefit from moving to the cloud is a unified medical records database, also known as electronic health record (EHR). In such an application, several hospitals, clinics, laboratories and public offices share patient records in order to offer a better service without the complexities of transferring patient information between them. A system like this was deployed in the UK for some years [Ehs]. Similarly to our previous example, availability of data is a fundamental requirement of a cloud-based EHR system, and privacy concerns are even more important.

A somewhat related example comes from the observation that some biomedical companies that generate high-value data would not put it on a third party cloud without ensuring confidentiality. In fact, some of these companies are actively stripping biomedical data stored on several clouds to avoid complete confidentiality loss in case of cloud compromise [May 2010].

All these applications can benefit from a system like DEPSKY. First, the fact that the information is replicated on several clouds would improve the data availability and integrity. Moreover, the DEPSKY-CA protocol (see Section 3.5) ensures the confidentiality of stored data and therefore addresses some of the privacy issues so important for these applications. Finally, these applications are prime examples of cases in which the extra costs due to replication are affordable for the added quality of service since the amount of data stored is not large when compared with Internet-scale services.

Content distribution. One of the most surprising uses of Amazon S3 is content distribution [Henry 2009]. In this scenario, users use the storage system as distribution points for their data in such a way that one or more producers store the content on their account and a set of consumers read this content. A system like DEPSKY that supports dependable updatable information storage can help this kind of application when the content being distributed is dynamic and there are security concerns associated. For example, a company can use the system to give detailed information about its business (price, available stock, etc.) to its affiliates with improved availability and security.

Future applications. Many applications are moving to the cloud, so, it is possible to think of new applications that would use the storage cloud as a back-end storage layer. Relational databases [Brantner et al. 2008], file systems [Vrable et al. 2012], objects stores and key-value databases are examples of systems that can use the cloud as storage layer as long as caching and weak consistency models [Terry et al. 1994; Vogels 2009] are used to avoid paying the price of cloud access in terms of latency and monetary costs per operation.

3. THE DEPSKY SYSTEM

As mentioned before, the clouds are storage clouds *without the capacity of executing users' code*, so they are accessed using their standard interface without modifications. The DEPSKY algorithms are implemented as a software library in the clients. This library offers an *object store* interface [Gibson et al. 1998], similar to what is used by parallel file systems (e.g., [Ghemawat et al. 2003; Weil et al. 2006]), allowing reads and writes in the back-end (in this case, the untrusted clouds). Figure 1 presents the architecture of DEPSKY.

In the remaining of this section we present our data and system models, the protocol design rationale, the two main protocols (DEPSKY-A and DEPSKY-CA), and some optimizations.

3.1. Data Model

The use of diverse clouds requires the DEPSKY library to deal with the heterogeneity of the interfaces of different cloud providers. An aspect that is specially important is the format of the data accepted by each cloud. The data model allows us to ignore these details when presenting the algorithms.

Figure 2 presents the DEPSKY data model with its three abstraction levels. In the first (left), there is the *conceptual data unit*, which corresponds to the basic storage object with which the *algorithms* work (a register in distributed computing parlance [Lamport 1986; Malkhi and Reiter 1998a]). A data unit has a unique name (X in the figure), a version number (to support updates on the object), verification data (usually a cryptographic hash of the data) and the data stored on the data unit object. In the second level (middle), the conceptual data unit is implemented as a *generic data unit* in an

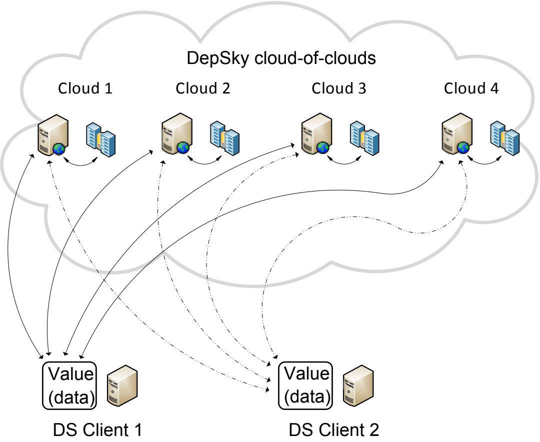


Fig. 1. Architecture of DEPSKY (with 4 clouds and 2 clients).

abstract storage cloud. Each generic data unit, or *container*, contains two types of files: a signed metadata file and the files that store the data. Metadata files contain the version number and the verification data, together with other information that applications may demand. Notice that a data unit (conceptual or generic) can store several versions of the data, i.e., the container can contain several data files. The name of the metadata file is simply *metadata*, while the data files are called *value-*⟨*Version*⟩, where ⟨*Version*⟩ is the version number of the data (e.g., *value-1*, *value-2*, etc.). Finally, in the third level (right) there is the *data unit implementation*, i.e., the container translated into the specific constructions supported by each cloud provider (Bucket, Folder, etc.). Notice that the one-container-per-data-unit policy may be difficult to implement in some clouds (e.g., Amazon S3 has a limit of 100 buckets per account, limiting the system to 100 data units). However, it is possible to store several data units on the same container as long as the data unit name is used as a prefix of their files names.

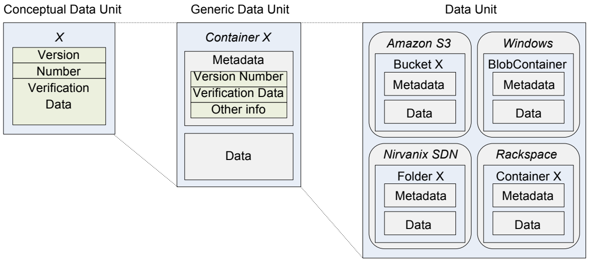


Fig. 2. DEPSKY data unit and the 3 abstraction levels.

The data stored on a data unit can have arbitrary size, and this size can be different for different versions. Each data unit object supports the usual object store operations: creation (create the container and the metadata file with version 0), destruction (delete or remove access to the data unit), write and read.

3.2. System Model

We consider an *asynchronous distributed system* composed of three types of parties: writers, readers and cloud storage providers. The latter are the clouds 1-4 in Figure 1, while writers and readers are roles of the clients, not necessarily different processes.

Readers and writers. Readers can fail arbitrarily, i.e., they can crash, fail intermittently and present any behavior. Writers, on the other hand, are assumed to fail only by crashing. We do not consider that writers can fail arbitrarily because, even if the protocol tolerated inconsistent writes in the replicas, faulty writers would still be able to write wrong values in data units, effectively corrupting the state of the application that uses DEPSKY. Moreover, the protocols that tolerate malicious writers are much more complex (e.g., [Cachin and Tessaro 2006; Liskov and Rodrigues 2006]), with active servers verifying the consistency of writer messages, which cannot be implemented on general storage clouds (Section 3.3).

All writers of a data unit *du* share a common private key $K_{r_w}^{du}$ used to sign some of the data written on the data unit (function $sign(DATA, K_{r_w}^{du})$), while readers of *du* have access to the corresponding public key $K_{u_w}^{du}$ to verify these signatures (function $verify(DATA, K_{u_w}^{du})$). This public key can be made available to the readers through the storage clouds themselves. Moreover, we assume also the existence of a collision-resistant *cryptographic hash function* H.

Cloud storage providers. Each cloud is modeled as a *passive storage entity* that supports five operations: *list* (lists the files of a container in the cloud), *get* (reads a file), *create* (creates a container), *put* (writes or modifies a file in a container) and *remove* (deletes a file). By passive storage entity, we mean that no protocol code other than what is needed to support the aforementioned operations is executed. We assume that access control is provided by the clouds in order to ensure that readers are only allowed to invoke the list and get operations (more about it in Section 5).

Since we do not trust clouds individually, we assume they can fail in a Byzantine way [Lamport et al. 1982]: stored data can be deleted, corrupted, created or leaked to unauthorized parties. This is the most general fault model and encompasses both malicious attacks/intrusions on a cloud provider and arbitrary data corruption (e.g., due to accidental events like the Ma.gnolia case). The protocols require a set of $n = 3f + 1$ storage clouds, at most f of which can be faulty. Additionally, the quorums used in the protocols are composed of any subset of $n - f$ storage clouds. It is worth to notice that this is the minimum number of replicas to tolerate Byzantine servers in asynchronous storage systems [Martin et al. 2002].

Readers, writers and clouds are said to be *correct* if they do not fail.

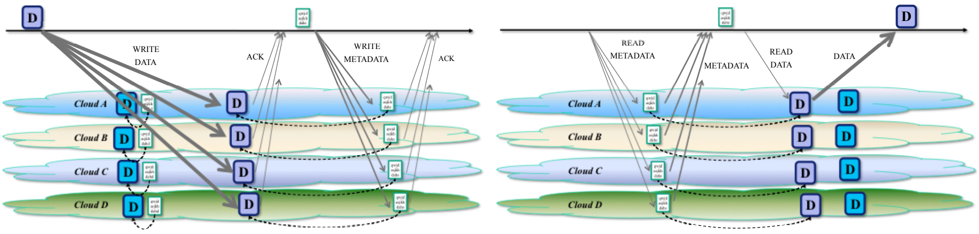
The register abstraction provided by DEPSKY satisfies a semantics that depends on the semantics provided by the underlying clouds. For instance, if the n clouds provide regular semantics, then DEPSKY also satisfies regular semantics: a read operation that happens concurrently with a write can return the value being written or the object's value before the write [Lamport 1986]. We discuss the semantics of DEPSKY in detail in Section 6.

Notice that our model hides most of the complexity of the distributed storage system employed by the cloud provider: it just assumes that this system is an object storage prone to Byzantine failures that supports very simple operations. These operations are accessed through RPCs (Remote Procedure Calls) with the following failure semantics: the operation keeps being invoked until a reply is received or the operation is canceled (possibly by another thread, using the *cancel_pending* operation to stop request retransmissions). This means that we have at most once semantics for the operations being invoked. Repeating the operation is not a problem because all storage cloud operations are idempotent, i.e., the state of the cloud becomes the same irrespectively of the operation being executed only once or more times.

3.3. Protocol Design Rationale

Quorum protocols can serve as the backbone of highly available storage systems [Chockler et al. 2009]. There are many quorum protocols for implementing Byzantine fault-tolerant (BFT) storage

[Cachin and Tessaro 2006; Goodson et al. 2004; Hendricks et al. 2007; Liskov and Rodrigues 2006; Malkhi and Reiter 1998a; Malkhi and Reiter 1998b; Martin et al. 2002], but most of them require that the servers execute protocol-specific code, a functionality not available on storage clouds. In consequence, cloud-specific protocols need to assume *passive storage replicas*, supporting only (blind) reads and writes. This leads to a key difference between the DEPSKY protocols and these classical BFT quorum protocols: *metadata and data are written and read in separate quorum accesses*. Moreover, these two accesses occur in different orders on read and write protocols, as depicted in Figure 3. This feature is crucial for the protocol correctness and efficiency.



(a) Write. (b) Read.

Fig. 3. DEPSKY read and write protocols.

Supporting multiple writers for a register (a data unit in DEPSKY parlance) can be problematic due to the lack of server code able to verify the version number of the data being written. To overcome this limitation we implement a single-writer multi-reader register, which is sufficient for many applications, and we provide a lock/lease protocol to support several concurrent writers for the data unit.

There are also some quorum protocols that consider individual storage nodes as passive shared memory objects (or disks) instead of servers [Abraham et al. 2006; Attiya and Bar-Or 2003; Chockler and Malkhi 2002; Gafni and Lamport 2003; Jayanti et al. 1998]. Unfortunately, most of these protocols require many steps to access the shared memory, or are heavily influenced by contention, which makes them impractical for geographically dispersed distributed systems such as DEPSKY due to the highly variable latencies involved. As show in Figure 3, DEPSKY protocols require two communication round-trips to read or write the metadata and the data files that are part of the data unit, independently of the existence of faults and contention.

Furthermore, as will be discussed later, many clouds do not provide the expected consistency guarantees of a disk, something that can affect the correctness of these protocols. The DEPSKY protocols provide *consistency-proportional semantics*, i.e., the semantics of a data unit is as strong as the underling clouds allow, from eventual to regular consistency semantics. We do not try to provide atomic (linearizable) semantics [Lamport 1986; Herlihy and Wing 1990] due to the fact that all known techniques require server-to-server communication [Cachin and Tessaro 2006], servers sending update notifications to clients [Martin et al. 2002] or write-backs [Goodson et al. 2004; Malkhi and Reiter 1998b; Basescu et al. 2012]. The first two mechanisms are not implementable using general purpose storage clouds (i.e., passive storage), while the last requires giving readers permission to write, nullifying our access control model.

To ensure the confidentiality of the data stored in the clouds we encrypt it using symmetric cryptography. To avoid the need of a key distribution service, which would have to be implement outside of the clouds, we employ a *secret sharing scheme* [Shamir 1979]. In this scheme, a dealer (the writer in the case of DEPSKY) distributes a secret (the encryption key) to n players (clouds in our case), but each player gets only a share of this secret. The main properties of the scheme is that at least $f+1 \leq n-f$ different shares of the secret are needed to recover it and that no information about the secret is disclosed with f or less shares. The scheme is integrated on the basic replication protocol in such way that each cloud stores just a share of the key used to encrypt the data being written. This ensures that no individual cloud will have access to the encryption key. On the contrary, clients that have authorization to access the data will be granted access to the key shares of (at least) $f+1$ different clouds, so they will be able to rebuild the encryption key and decrypt the data.

The use of a secret sharing scheme allows us to integrate confidentiality guarantees to the stored data without using a key distribution mechanism to make writers and readers of a data unit share a secret key. In fact, our mechanism reuses the access control of the cloud provider to control which readers are able to access the data stored on a data unit.

Although it may seem questionable if avoiding key distribution methods is useful for a large spectrum of applications, our previous experience with secret sharing schemes [Bessani et al. 2008] suggests that the overhead of using them is not deterrent, specially if one considers the communication latency of accessing a cloud storage provider. Nevertheless, the protocol can be easily modified to use a shared key for confidentiality if an external key distribution method is available.

If we simply replicate the data on n clouds, the monetary costs of storing data using DEPSKY would increase by a factor of n. In order to avoid this, we compose the secret sharing scheme used on the protocol with an *information-optimal erasure code algorithm*, reducing the size of each share by a factor of $\frac{n}{f+1}$ of the original data [Rabin 1989]. This composition follows the original proposal of [Krawczyk 1993], where the data is encrypted with a random secret key, the encrypted data is encoded, the key is divided using secret sharing and each server receives a block of the encrypted data and a share of the key.

Common sense says that for critical data it is always a good practice not erasing all old versions of the data, unless we can be certain that we will not need them anymore [Hamilton 2007]. An additional feature of our protocols is that old versions of the data are kept in the clouds unless they are explicitly deleted.

3.4. DEPSKY-A – Available DepSky

The first DEPSKY protocol is called DEPSKY-A. It improves the availability and integrity of cloud-stored data by replicating it on several clouds using quorum techniques. Algorithm 1 presents this protocol. We encapsulate some of the protocol steps in the functions described in Table I. We use the '.' operator to denote access to metadata fields, e.g., given a metadata file m, $m.ver$ and $m.digest$ denote the version number and digest(s) stored in m. We use the '+' operator to concatenate two items into a string, e.g., "value-"+*new_ver* produces a string that starts with the string "value-" and ends with the value of variable *new_ver* in string format. Finally, the *max* function returns the maximum among a set of numbers.

Table I. Functions used in the DEPSKY-A protocols (implementation in Appendix A).

Function	Description
$queryMetadata(du)$	Obtains the correctly signed file metadata stored in the container du of $n-f$ clouds used to store the data unit and returns it in an array.
$writeQuorum(du,\ name, value)$	For every cloud $i \in \{0,...,n-1\}$, writes the $value[i]$ on a file named $name$ on the container du in that cloud and waits for write confirmations from $n-f$ clouds.

The key idea of the *write algorithm* (lines 1-13) is to first write the value in a quorum of clouds (line 8), then write the corresponding metadata (line 12), as illustrated in Figure 3(a). This order of operations ensures that a reader will only be able to read metadata for a value already stored in the clouds. Additionally, when a writer first writes a data unit *du* (lines 3-5, max_ver_{du} initialized with 0), it first contacts the clouds to obtain the metadata with the greatest version number, then updates the max_ver_{du} variable with the current version of the data unit.

The *read algorithm* starts by fetching the metadata files from a quorum of clouds (line 16) and choosing the one with greatest version number (line 17). After that, the algorithm enters in a loop where it keeps looking at the clouds until it finds the data unit version corresponding to this version number and the cryptographic hash found in the chosen metadata (lines 18-26). Inside of this loop,

ALGORITHM 1: DEPSKY-A read and write protocols.

```
1  procedure DepSkyAWrite(du,value)
2  begin
3      if $max\_ver_{du} = 0$ then
4          $m \longleftarrow queryMetadata(du)$
5          $max\_ver_{du} \longleftarrow \max(\{m[i].ver \ : \ 0 \leq i \leq n-1\})$
6      $new\_ver \longleftarrow max\_ver_{du} + 1$
7      $v[0 \,..\, n-1] \longleftarrow value$
8      $writeQuorum(du,$"value-"$+new\_ver, v)$
9      $new\_meta \longleftarrow \langle new\_ver, \mathsf{H}(value)\rangle$
10     $sign(new\_meta, K^{du}_{r_w})$
11     $v[0 \,..\, n-1] \longleftarrow new\_meta$
12     $writeQuorum(du,$"metadata"$, v)$
13     $max\_ver_{du} \longleftarrow new\_ver$

14 function DepSkyARead(du)
15 begin
16     $m \longleftarrow queryMetadata(du)$
17     $max\_id \longleftarrow i \ : \ m[i].ver = \max(\{m[i].ver \ : \ 0 \leq i \leq n-1\})$
18     repeat
19         $v[0 \,..\, n-1] \longleftarrow \perp$
20         parallel for $0 \ \leq \ i \ < \ n-1$ do
21             $tmp_i \longleftarrow cloud_i.get(du,$"value-" $+m[max\_id].ver)$
22             if $\mathsf{H}(tmp_i) = m[max\_id].digest$ then $v[i] \longleftarrow tmp_i$
23             else $v[i] \longleftarrow ERROR$
24         wait until $(\exists i : v[i] \neq \perp \wedge v[i] \neq ERROR) \vee (|\{i : v[i] \neq \perp\}| \geq n-f)$
25         for $0 \ \leq \ i \ \leq \ n-1$ do $cloud_i.cancel\_pending()$
26     until $\exists i : v[i] \neq \perp \wedge v[i] \neq ERROR$
27     return $v[i]$
```

the process fetches the file from the clouds until either it finds one value file containing the value matching the digest on the metadata or the value is not found on at least $n-f$ clouds[1] (lines 20-24). Finally, when a valid value is read, the reader cancels the pending RPCs, exits the loop and returns the value (lines 25-27). The normal case execution (with some optimizations discussed in Section 3.6) is illustrated in Figure 3(b).

The rationale of why this protocol provides the desired properties is the following (proofs in the Appendix B). Availability is guaranteed because the data is stored in a quorum of at least $n-f$ clouds and it is assumed that at most f clouds can be faulty. The read operation has to retrieve the value from only one of the clouds (line 22), which is always available because $(n-f)-f > 1$. Together with the data, signed metadata containing its cryptographic hash is also stored. Therefore, if a cloud is faulty and corrupts the data, this is detected when the metadata is retrieved. Moreover, the fact that metadata files are self-verifiable (i.e., signed) and quorums overlap in at least $f+1$ clouds (one correct) ensures the last written metadata file will be read. Finally, the outer loop of the read ensures that the read of a value described on a read metadata will be repeated until it is available, which will eventually hold since a metadata file is written only after the data file is written.

3.5. DEPSKY-CA – Confidential & Available DepSky

The DEPSKY-A protocol has two main limitations. First, a data unit of size S consumes $n \times S$ storage capacity of the system and costs on average n times more than if it was stored in a single

[1] This is required to avoid the process to block forever waiting replies from f faulty clouds.

cloud. Second, it stores the data in cleartext, so it does not give confidentiality guarantees. To cope with these limitations we employ an information-efficient secret sharing scheme [Krawczyk 1993] that combines symmetric encryption with a classical secret sharing scheme and an optimal erasure code to partition the data in a set of blocks in such a way that (i.) $f+1$ blocks are necessary to recover the original data and (ii.) f or less blocks do not give any information about the stored data[2]. The overall process is illustrated in Figure 4.

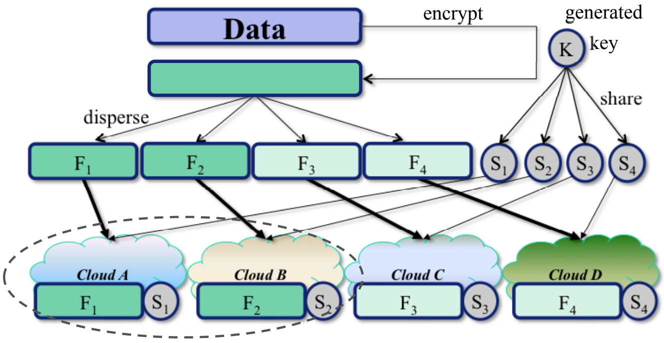


Fig. 4. The combination of symmetric encryption, secret sharing and erasure codes in DEPSKY-CA.

The DEPSKY-CA protocol integrates these techniques with the DEPSKY-A protocol (Algorithm 2). The additional cryptographic and coding functions needed are in Table II. The differences of the DEPSKY-CA protocol in relation to DEPSKY-A are the following: (1.) the encryption of the data, the generation of the key shares and the encoding of the encrypted data on DepSkyCAWrite (lines 7-10) and the reverse process on DepSkyCARead (lines 33-35), as shown in Figure 4; (2.) the data stored in $cloud_i$ is composed of the share of the key $s[i]$ and the encoded block $v[i]$ (line 12); and (3.) $f+1$ replies are necessary to read the data unit's current value instead of one on DEPSKY-A (lines 30 and 32). Additionally, instead of storing a single digest on the metadata file, the writer generates and stores n digests, one for each cloud. These digests are accessed as different positions of a vector stored in the *digest* field of a metadata. If a key distribution infrastructure is available, or if readers and writers share a common key k for each data unit, the secret sharing scheme can be removed (lines 7, 9 and 34 are not necessary).

The rationale of the correctness of the protocol is similar to the one for DEPSKY-A (proofs also in the Appendix B). The main differences are those already pointed out: encryption prevents individual clouds from disclosing the data; secret sharing allows storing the encryption key in the cloud without f faulty clouds being able to reconstruct it; the erasure code scheme reduces the size of the data stored in each cloud.

3.6. Optimizations

This section introduces two optimizations that can make the protocols more efficient and cost-effective. In Section 8 we evaluate the impact of these optimizations on the protocols.

Write. In the DEPSKY-A and DEPSKY-CA write algorithms, a value file is written using the function *writeQuorum* (see Table I). This function tries to write the file on all clouds and waits for confirmation from a quorum. A more cost-effective solution would be to try to store the value only on a *preferred quorum*, resorting on extra clouds only if the reception of write confirmations from the quorum of clouds is not received until a timeout. This optimization can be applied both to DEPSKY-A and DEPSKY-CA to make the data be stored only in $n-f$ out-of n clouds, which

[2] Erasure codes alone cannot satisfy this confidentiality guarantee.

ALGORITHM 2: DEPSKY-CA read and write protocols.

```
procedure DepSkyCAWrite(du,value)
begin
    if $max\_ver_{du} = 0$ then
        $m \longleftarrow queryMetadata(du)$
        $max\_ver_{du} \longleftarrow \max(\{m[i].version \ : \ 0 \le i \le n-1\})$
    $new\_ver \longleftarrow max\_ver_{du} + 1$
    $k \longleftarrow generateSecretKey()$
    $e \longleftarrow \mathsf{E}(value, k)$
    $s[0 \,..\, n-1] \longleftarrow share(k, n, f+1)$
    $v[0 \,..\, n-1] \longleftarrow encode(e, n, f+1)$
    for $0 \ \le \ i \ < \ n-1$ do
        $d[i] \longleftarrow \langle s[i], v[i] \rangle$
        $h[i] \longleftarrow \mathsf{H}(d[i])$
    $writeQuorum(du,$"value-"$+new\_ver, d)$
    $new\_meta \longleftarrow \langle new\_ver, h \rangle$
    $sign(new\_meta, K_{r_w}^{du})$
    $v[0 \,..\, n-1] \longleftarrow new\_meta$
    $writeQuorum(du,$"metadata"$, v)$
    $max\_ver_{du} \longleftarrow new\_ver$

function DepSkyCARead(du)
begin
    $m \longleftarrow queryMetadata(du)$
    $max\_id \longleftarrow i \ : \ m[i].ver \ = \max(\{m[i].ver \ : \ 0 \le i \le n-1\})$
    repeat
        $d[0 \,..\, n-1] \longleftarrow \perp$
        parallel for $0 \ \le \ i \ \le \ n-1$ do
            $tmp_i \longleftarrow cloud_i.get(du,$ "value-" $+m[max\_id].ver)$
            if $\mathsf{H}(tmp_i) = m[max\_id].digest[i]$ then $d[i] \longleftarrow tmp_i$
            else $d[i] \longleftarrow ERROR$
        wait until $(|\{i \ : \ d[i] \ne \perp \wedge d[i] \ne ERROR\}| > f) \vee (|\{i \ : \ d[i] \ne \perp\}| > n-f)$
        for $0 \ \le \ i \ \le \ n-1$ do $cloud_i.cancel\_pending()$
    until $|\{i \ : \ d[i] \ne \perp \wedge d[i] \ne ERROR\}| > f$
    $e \longleftarrow decode(d.e, n, f+1)$
    $k \longleftarrow combine(d.s, n, f+1)$
    return $\mathsf{D}(e, k)$
```

Table II. Functions used in the DEPSKY-CA protocols.

Function	Description
$generateSecretKey()$	Generates a random secret key.
$\mathsf{E}(v,k)$/$\mathsf{D}(e,k)$	Encrypts v and decrypts e with key k.
$encode(d,n,t)$	Encodes d on n blocks in such a way that t are required to recover it.
$decode(db,n,t)$	Decodes array db of n blocks, with at least t valid, to recover d.
$share(s,n,t)$	Generates n shares of s in such a way that at least t of them are required to obtain any information about s.
$combine(ss,n,t)$	Combines shares on array ss of size n containing at least t correct shares to obtain the secret s.

can decrease the DEPSKY storage cost by a factor of $\frac{n-f}{n}$, possibly with some loss in terms of availability and durability of the data.

Read. The DEPSKY-A algorithm described in Section 3.4 tries to read the most recent version of the data unit from all clouds and waits for the first valid reply to return it. In the pay-per-use model this is far from ideal because the user will pay for n data accesses. A lower-cost solution is to use some criteria to sort the clouds and try to access them sequentially, one at time, until the value is obtained. The sorting criteria can be based on access monetary cost (cost-optimal), the latency of *queryMetadata* on the protocol (latency-optimal), a mix of the two or any other more complex criteria (e.g., a history of the latency and faults of the clouds).

This optimization can also be used to decrease the monetary cost of the DEPSKY-CA read operation. The main difference is that instead of choosing one of the clouds at a time to read the data, $f+1$ of them are chosen.

4. DEPSKY EXTENSIONS

In this section we present a set of additional protocols that may be useful for implementing real systems using DEPSKY.

4.1. Supporting Multiple Writers – Locking with Storage Clouds

The DEPSKY protocols presented do not support concurrent writes, which is sufficient for many applications where each process writes on its own data units. However, there are applications in which this is not the case. An example is a fault-tolerant storage system that uses DEPSKY as its back-end object store. This system could have more than one node with the writer role writing in the same data unit(s) for fault tolerance reasons. If the writers are in the same network, coordination services like ZooKeeper [Hunt et al. 2010] or DepSpace [Bessani et al. 2008] can be used to elect a leader and coordinate the writes. However, if the writers are scattered through the Internet this solution is not practical without trusting the site in which the coordination service is deployed (and even in this case, the coordination service may be unavailable due to network issues). Open coordination services such as WSDS [Alchieri et al. 2008] can still be used, but they require an Internet deployment.

The solution we advocate is a *low contention lock mechanism* that uses the cloud-of-clouds itself to maintain lock files on a data unit. These files specify which is the writer and for how much time it has write access to the data unit. However, for this solution to work, two additional assumptions must hold. The first one is related with the use of leases. The algorithm requires every contending writer to have *synchronized clocks* with a precision of Δ. This can be ensured in practice by making all writers that want to lock a data unit synchronize their clocks with a common NTP (Network Time Protocol [Mills 1992]) server with a precision of $\frac{\Delta}{2}$. The second assumption is related with the consistency of the clouds. We assume *regular semantics* [Lamport 1986] for the creation and listing of files on a container. Although this assumption appears to be too strong, object storage services like Amazon S3 already ensure this kind of consistency for object creation, sometimes called *read-after-write* [Amazon Web Services 2011]. Anyway, in Section 6 we discuss the effects of weakly consistent clouds on this protocol.

The lock protocol is described in Algorithm 3, and it works as follows. A process c that wants to be a writer (and has permission to be), first lists files on the data unit container on a quorum of clouds and tries to find a valid file called *lock-c'-T'* with $c' \neq c$ and local time on the process smaller than $T' + \Delta$ (lines 5-10). If such file is found in some cloud, it means that some other process c' holds the lock for this data unit and c will sleep for a random amount of time before trying to acquire the lock again (line 21). If the file is not found, c can write a lock file named *lock-c-T* containing a digital signature of the file name on all clouds (lines 11 and 12), being $T = local\_clock + LEASE\_TIME$. In the last step, c lists again all files in the data unit container searching for valid and not expired lock files from other processes (lines 13-17). If a file like that is found, c removes the lock file it wrote from the clouds and sleeps for a small random amount of time before trying to run the protocol again (lines 18-21). Otherwise, c becomes the single-writer for the data unit until T.

The protocol also uses a predicate *valid* that verifies if the lock file was not created by a faulty cloud. The predicate is true if the lock file is either returned by $f+1$ clouds or its contents is correctly signed by its owner (line 28).

ALGORITHM 3: DepSky data unit locking by writer c.

```
1  function DepSkyLock(du)
2  begin
3      $lock\_id \leftarrow \perp$
4      repeat
           // list lock files on all clouds to see if the du is locked
5          $L[0 .. n-1] \longleftarrow \perp$
6          parallel for $0 \leq i \leq n-1$ do
7              $L[i] \longleftarrow cloud_i.list(du)$
8          wait until $(|\{i : L[i] \neq \perp\}| > n-f)$
9          for $0 \leq i \leq n-1$ do $cloud_i.cancel\_pending()$
10         if $\nexists i : \exists lock\text{-}c'\text{-}T' \in L[i] : c' \neq c \wedge valid(L, lock\text{-}c'\text{-}T', du) \wedge (T' + \Delta > local\_clock)$ then
               // create a lock file for the du and write it in the clouds
11             $lock\_id \leftarrow$ "lock-"$+c+$"-"$+(local\_clock + LEASE\_TIME)$
12             $writeQuorum(du, lock\_id, sign(lock\_id, K^{du}_{r_c}))$
               // list the lock files again to detect contention
13             $L[0 .. n-1] \longleftarrow \perp$
14             parallel for $0 \leq i \leq n-1$ do
15                 $L[i] \longleftarrow cloud_i.list(du)$
16             wait until $(|\{i : L[i] \neq \perp\}| > n-f)$
17             parallel for $0 \leq i \leq n-1$ do $cloud_i.cancel\_pending()$
18             if $\exists i : \exists lock\text{-}c'\text{-}T' \in L[i] : c' \neq c \wedge valid(L, lock\text{-}c'\text{-}T', du) \wedge (T' + \Delta > local\_clock)$ then
19                 $DepSkyUnlock(lock\_id)$
20                 $lock\_id \leftarrow \perp$
21         if $lock\_id = \perp$ then sleep for some time
22     until $lock\_id \neq \perp$
23     return $lock\_id$

24 procedure DepSkyUnlock(lock_id)
25 begin
26     parallel for $0 \leq i < n-1$ do
27         $cloud_i.delete(du, \text{lock\_id})$

28 predicate $valid(L, \text{lock-c'-T'}, du) \equiv (|\{i : \text{lock-c'-T'} \in L[i]\}| > f \vee verify(\text{lock-c'-T'}, K^{du}_{u_c}))$
```

Several remarks can be made about this protocol. First, the backoff strategy is necessary to ensure that two processes trying to become writers at the same time never succeed. Second, locks can be renewed periodically to ensure existence of a single writer at every moment of the execution. Unlocking can be easily done through the removal of the lock files (lines 24-27). Third, this lock protocol is only *obstruction-free* [Herlihy et al. 2003]: if several process try to become writers at the same time, it is possible that none of them is successful. However, due to the backoff strategy used, this situation should be very rare on the envisioned deployments. Finally, it is important to notice that the unlock procedure is not fault-tolerant: in order to release a lock, the lock file has to be deleted from all clouds; a malicious cloud can still show the removed lock file disallowing lock acquisition by other writers. However, given the finite validity of a lock, this problem can only affect the system for a limited period of time, after which the problematic lock expires.

The proof that this protocol satisfies mutual exclusion and obstruction-freedom is presented in Appendix C.

4.2. Management Operations

Besides read, write and lock, DEPSKY provides other operations to manage data units. These operations and underlying protocols are briefly described in this section.

Creation and destruction. Creating a data unit can be easily accomplished by invoking the create operation in each individual cloud. In contention-prone applications, the creator should execute the locking protocol of the previous section before executing the first write to ensure it is the single writer of the data unit.

The destruction of a data unit is done in a similar way: the writer simply removes all files and the container that stores the data unit by calling *remove* in each individual cloud.

Garbage collection. As already discussed in Section 3.3, we choose to keep old versions of the value of the data unit on the clouds to improve the dependability of the storage system. However, after many writes the amount of storage used by a data unit can become very high and thus some garbage collection is necessary. The protocol for doing that is very simple: a writer just lists all files named "value-*version*" in the data unit container and removes all those with *version* smaller than the oldest version it wants to keep in the system.

Cloud reconfiguration. Sometimes one cloud can become too expensive or too unreliable to be used for storing DEPSKY data units. For such cases DEPSKY provides a reconfiguration protocol that substitutes one cloud by another. The protocol is the following: (1.) the writer reads the data (probably from the other clouds and not from the one being removed); (2.) creates the data unit container on the new cloud; (3.) executes the write protocol on the clouds not removed and the new cloud; (4.) deletes the data unit from the cloud being removed. After that, the writer needs to inform the readers that the data unit location was changed. This can be done writing a special file on the data unit container of the remaining clouds informing the new configuration of the system. A process will accept the reconfiguration if this file is read from at least $f+1$ clouds. Notice that this protocol only works if there are no writes being executed during the reconfiguration, which might implies the use of the locking protocol described in the previous subsection if the data unit has multiple writers.

5. CLOUD-OF-CLOUDS ACCESS CONTROL

In this section we briefly discuss how cloud storage access control can be used to set up the access control for management, writers and readers of DEPSKY data units.

Management. The management operations described in Section 4.2 can only be executed by writers of a data unit, with the exception of the creation and destruction of a data unit, that needs to be carried on by the data unit's owner, that has write rights on the data unit container parent directory.

Writers. If a data unit has more than one possible writer, all of them should have the write rights on the data unit container. Moreover, all writers first write their public keys on the DU container before trying to acquire the lock for writing on the data unit. Notice that it is possible to have a single writer account, with a single shared writer private and public key pair, being used by several writer processes (e.g., for fault tolerance reasons). Finally, when a writer does not need to write in a data unit anymore, it removes its public key from the data unit container on all clouds.

Readers. The readers of a data unit are defined by the set of accounts that have read access to the data unit container. It is worth to mention that some clouds such as Rackspace Files and Nirvanix CDN do not provide this kind of rich access control. These clouds only allow a file to be confidential (accessed only by its writer) or public (accessed by everyone that knows its URL). However, other popular storage clouds like Amazon S3, Windows Azure Blob Service and Google Drive support

ACLs for giving read (and write) access to the files stored in a single account. We expect this kind of functionality to be available in most storage clouds in the near future.

Finally, all readers of a data unit consider that a metadata or lock file is correctly signed if the signature was produced with any of the writer keys available on the data unit container of $f+1$ clouds.

6. CONSISTENCY PROPORTIONALITY

Both DEPSKY-A and DEPSKY-CA protocols implement *single-writer multi-reader regular registers* if the clouds being accessed provide *regular semantics* [Lamport 1986]. However, several clouds do not guarantee this semantics, but instead provide *read-after-write* (which is similar to the *safe semantics* [Lamport 1986]) or *eventual consistency* [Vogels 2009] for the data stored (e.g., Amazon S3 [Ama]).

In fact, the DEPSKY read and write protocols are *consistency-proportional* in the following sense: *if the underlying clouds support a consistency model $\mathscr{C}$, the* DEPSKY *protocols provide consistency model $\mathscr{C}$*. This holds for any $\mathscr{C}$ among the following: *eventual* [Vogels 2009], *read-your-writes*, *monotonic reads*, *writes-follow-reads*, *monotonic writes* [Terry et al. 1994] and *read-after-write* [Lamport 1986]. These models are briefly described in Table III. A proof that DEPSKY provides consistency proportionality can be found in Appendix D.

Table III. Consistency models supported by DEPSKY.

Consistency Model	Brief Description
Eventual [Vogels 2009]	Written values will be eventually reflected in read operations.
Read-your-writes [Terry et al. 1994]	Read operations reflect previous writes.
Monotonic reads [Terry et al. 1994]	Successive reads reflect a nondecreasing set of writes.
Writes-follow-reads [Terry et al. 1994]	Writes are propagated after reads on which they depend.
Monotonic writes [Terry et al. 1994]	Writes are propagated after writes that logically precede them.
Read-after-write [Lamport 1986]	After a write completes, it will be reflected in any posterior read.

Notice that if the underlying clouds are heterogeneous in terms of consistency guarantees, DEPSKY provides the weakest consistency among those provided. This comes from the fact that the consistency of a read directly depends of the reading of the last written metadata file. Since we use read and write quorums with at least $f+1$ clouds in their intersections, and since at most f clouds may be faulty, the read of the most recently written metadata file may happen in the single correct cloud in such intersection. If this cloud does not provide strong consistency, the whole operation will be weakly consistent, following the consistency model of this cloud.

A problem with not having regular consistent clouds is that the lock protocol may not work correctly. After listing the contents of a container and not seeing a file, a process cannot conclude that it is the only writer. This problem can be minimized if the process waits a while between lines 12 and 13 of Algorithm 3. However, the mutual exclusion guarantee will only be satisfied if the wait time is greater than the time for a data written to be seen by every other reader. Unfortunately, no eventually consistent cloud of our knowledge provides this kind of timeliness guarantee, but we can experimentally discover the amount of time needed for a read to propagate on a cloud with the desired coverage and use this value in the aforementioned wait. Moreover, to ensure some safety even when two writes happen in parallel, we can include a unique id of the writer (e.g., the hash of part of its private key) as the decimal part of its timestamps, just like it is done in most Byzantine quorum protocols (e.g., [Malkhi and Reiter 1998a]). This simple measure allows the durability of data written by concurrent writers (the name of the data files will be different), even if the metadata file may point to different versions on different clouds.

7. DEPSKY IMPLEMENTATION

We have implemented a DEPSKY prototype in Java as an application library that supports the read and write operations. The code is divided in three main parts: (1) data unit manager, that stores the definition and information of the data units that can be accessed; (2) system core, that implements the DEPSKY-A and DEPSKY-CA read and write protocols; and (3) cloud drivers, which implements the logic for accessing the different clouds. The current implementation has 5 drivers available (the four clouds used in the evaluation - see next section - and one for storing data locally), but new drivers can be easily added. The overall implementation comprises about 2900 lines of code, being 1100 lines for the drivers. The most recent version of the code is available at `http://code.google.com/p/depsky/`.

The DEPSKY code follows a model of one thread per cloud per data unit in such a way that the cloud accesses can be executed in parallel (as described in the algorithms). All communications between clients and cloud providers are made over HTTPS (secure and private channels) using the REST APIs supplied by the storage cloud providers. Some of the clouds are accessed using the libraries available from the providers. To avoid problems due to the differences in implementation, in particular with different retransmission timeouts and retry policies, we disabled this feature from the drivers and implemented it on our code. The result is that all clouds are accessed using the same timeout and number of retries in case of failure.

The prototype employs *speculation* to execute the two phases of the read protocols in parallel. More precisely, as soon as a metadata file is read from a cloud i, the system starts fetching the data file from i, without waiting for $n-f$ metadata to find the one with greatest version number. The idea is to minimize access latency (which varies significantly in the different clouds) under the assumption that contention between reads and writes is rare and Byzantine faults seldom happen.

Our implementation makes use of several building blocks: RSA with 1024 bit keys for signatures, SHA-1 for cryptographic hashes, AES for symmetric cryptography, Shoenmakers' PVSS scheme [Schoenmakers 1999] for secret sharing with 192 bits secrets and the classic Reed-Solomon for erasure codes [Plank 2007]. Most of the implementations used come from the Java 6 API, while Java Secret Sharing [Bessani et al. 2008] and Jerasure [Plank 2007] were used for secret sharing and erasure codes, respectively.

8. EVALUATION

In this section we present an evaluation of DEPSKY which tries to answer three main questions: *What is the additional cost in using replication on storage clouds? What are the advantages and drawbacks in terms of performance and availability of using multiple clouds to store data? What are the relative costs and benefits of the two* DEPSKY *protocols?*

The evaluation focus on the case of $n = 4$ and $f = 1$, which we expect to be the common deployment setup of our system for two reasons: (1.) f is the maximum number of faulty cloud storage providers, which are very resilient and so faults should be rare; (2.) there is additional complexity in setting up a great number of accounts in different clouds to be used in DEPSKY. Our evaluation uses the following cloud storage services with their default configurations: Amazon S3, Windows Azure Blob Store, Nirvanix CDN and Rackspace Files.

8.1. Monetary cost evaluation

Storage cloud providers usually charge their users based on the number of operations executed and the amount of data uploaded, downloaded and stored on them. Table IV presents the cost in US Dollars of executing 10000 reads and writes using the DEPSKY data model (with metadata and supporting many versions of a data unit) considering three data unit sizes: 100kB, 1MB and 10MB. This table includes only the costs of the operations being executed (invocations, upload and download), not the data storage, which will be discussed later. All estimations presented in this section were calculated based on the values charged by the four clouds at September 25th, 2010.

In the table, the columns "DS-A", "DS-A opt", "DS-CA" e "DS-CA opt" present the costs of using the DEPSKY protocols with the optimizations discussed in Section 3.6 disabled and enabled, respectively. The other columns present the costs for storing the data unit (DU) in a single cloud.

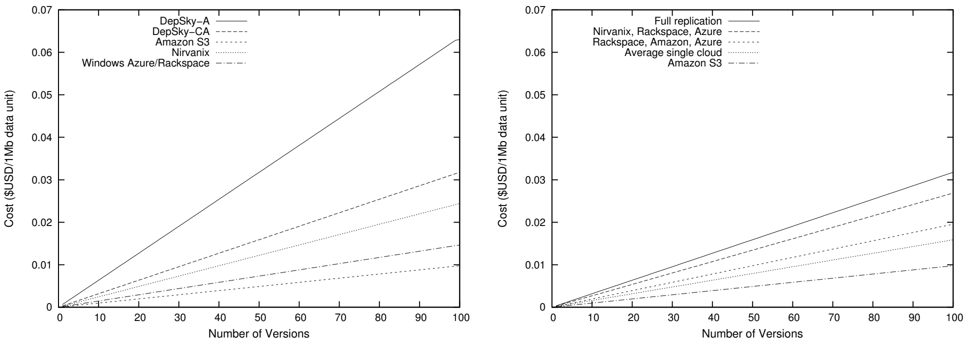
The table shows that the cost of DEPSKY-A with $n = 4$ and without optimizations is roughly the sum of the costs of using the four clouds, as expected. However, if the read optimization is

Table IV. Estimated costs per 10000 operations (in US Dollars). DepSky-A (DS-A) and DepSky-CA (DS-CA) costs are computed for the realistic case of 4 clouds ($f = 1$). The "DS-A opt" and "DS-CA opt" setups consider the cost-optimal version of the protocols with no failures.

Operation	DU	DS-A	DS-A opt	DS-CA	DS-CA opt	Amazon	Rackspace	Azure	Nirvanix
10000 Reads	100kB	0.64	0.14	0.32	0.14	0.14	0.21	0.14	0.14
	1MB	6.55	1.47	3.26	1.47	1.46	2.15	1.46	1.46
	10MB	65.5	14.6	32.0	14.6	14.6	21.5	14.6	14.6
10000 Writes	100kB	0.60	0.32	0.30	0.17	0.14	0.08	0.09	0.29
	1MB	6.16	3.22	3.08	1.66	1.46	0.78	0.98	2.93
	10MB	61.5	32.2	30.8	16.6	14.6	7.81	9.77	29.3

employed, the less expensive cloud cost dominates the cost of executing reads (only one out-of four clouds is accessed in fault-free executions). If the optimized write is employed, the data file will be written only on a preferred quorum excluding the most expensive cloud (Nirvanix), and thus the costs will be substantially smaller. For DepSky-CA, the cost of reading and writing without optimizations is approximately 50% of DepSky-A's due to the use of information-optimal erasure codes that make the data stored on each cloud roughly 50% of the size of the original data. The optimized version of DepSky-CA also reduces the read cost to half of the sum of the two less costly clouds due to its access to only $f+1$ clouds in the best case, while the write cost is reduced since Nirvanix is not used. Recall that the costs for the optimized versions of the protocol account only for the best case in terms of monetary costs: reads and writes are executed on the required less expensive clouds. In the worst case, the more expensive clouds will also be used.

The storage costs of a 1MB data unit for different numbers of stored versions is presented in Figure 5. We present the curves only for one data unit size because other size costs are directly proportional.



(a) Storage costs.

(b) DepSky-CA with preferred quorums.

Fig. 5. Storage costs of a 1MB data unit for different numbers of stored versions in different DepSky setups and clouds.

The results depicted in the Figure 5(a) show that the cost of DepSky-CA storage without employing preferred quorums is roughly half the cost of using DepSky-A and twice the cost of using a single cloud. This is no surprise since the storage costs are directly proportional to the amount of data stored on the cloud, and DepSky-A stores 4 times the data size, while DepSky-CA stores 2 times the data size and an individual cloud just stores a single copy of the data.

Figure 5(b) shows some results considering the case in which the data stored using DepSky-CA is stored only on a preferred quorum of clouds (see Section 3.6). The figure contains values for the less expensive preferred quorum (Amazon S3, Windows Azure and Rackspace) and the most expensive preferred quorum (Nirvanix, Windows Azure and Rackspace) together with Amazon S3 and DEPSKY-CA writing on all clouds for comparison. The results show that the use of preferred quorums decreases the storage costs between 15% (most expensive quorum) to 38% (less expensive quorum) when compared to the full replicated DEPSKY-CA. Moreover, in the best case, DEPSKY-CA can store data with an additional cost of only 23% more than the average cost to store data on a single cloud and twice the cost of the least expensive cloud (Amazon S3).

Notice that the metadata costs are almost irrelevant when compared with the data size since its size is less than 500 bytes.

8.2. Performance and availability evaluation

In order to understand the performance of DEPSKY in a real deployment, we used PlanetLab to run several clients accessing a cloud-of-clouds composed of popular storage cloud providers. This section explains our methodology and presents the obtained results in terms of read and write latency, throughput and availability.

Methodology. The latency measurements were obtained using a logger application that tries to read a data unit from six different clouds: the four storage clouds individually and the two clouds-of-clouds implemented with DEPSKY-A and DEPSKY-CA.

The logger application executes periodically a *measurement epoch*, which comprises: read the data unit (DU) from each of the clouds individually, one after another; read the DU using DEPSKY-A; read the DU using DEPSKY-CA; sleep until the next epoch. The goal is to read the data through different setups within a time period as small as possible in order to minimize the impact of Internet performance variations.

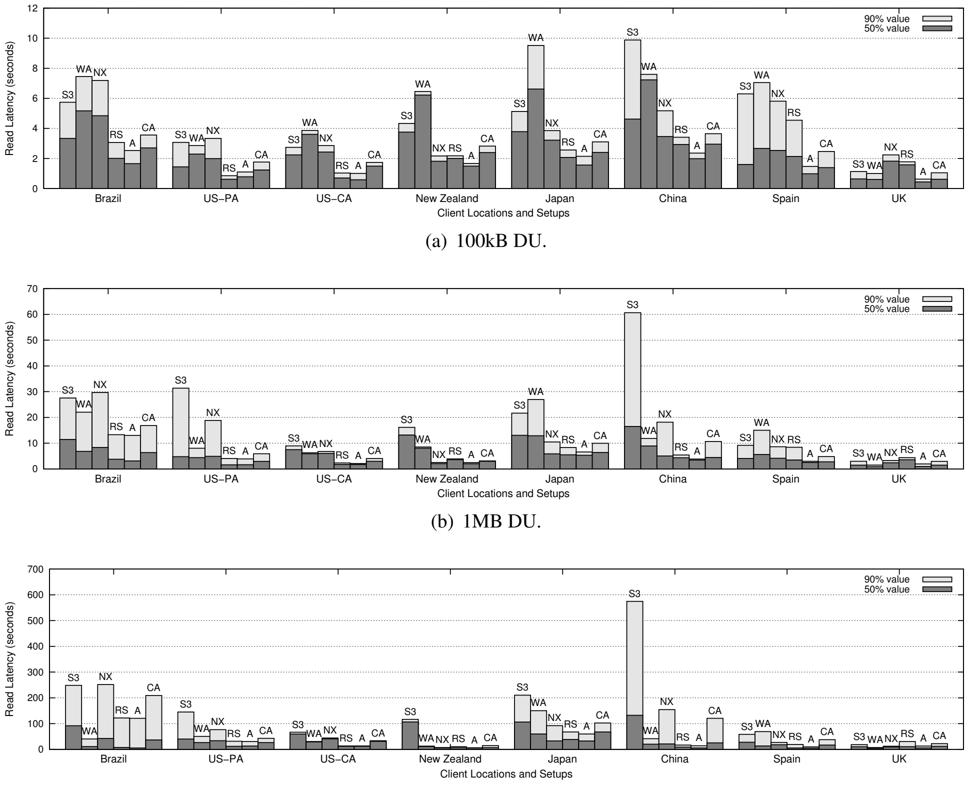
We deployed the logger on eight PlanetLab machines across the Internet, on five continents. In each of these machines three instances of the logger were started for different DU sizes: 100kB (a measurement every 5 minutes), 1MB (a measurement every 10 minutes) and 10MB (a measurement every 30 minutes). These experiments took place during two months, but the values reported correspond to measurements done between September 10, 2010 and October 7, 2010.

In the experiments, the local costs, which the protocols incur due to the use of cryptography and erasure codes, are negligible for DEPSKY-A and account for at most 5% of the read and 10% of the write latencies on DEPSKY-CA.

Reads. Figure 6 presents the 50% and 90% percentile of all observed latencies of the reads executed (i.e., the values in which 50% and 90% of the observations fell below). These experiments were executed without the (monetary) read optimization described in Section 3.6. The number of reads executed on each site is presented on the second column of Table VII.

Based on the results presented in the figure, several points can be highlighted. First, DEPSKY-A presents the best latency in all cases but one. This is explained by the fact that it waits for 3 out-of 4 copies of the metadata but only one of the data, and it usually obtains it from the best cloud available during the execution. Second, DEPSKY-CA's latency is closely related with the second best cloud storage provider, since it waits for at least 2 out-of 4 data blocks. Finally, there is a huge variance between the performance of the cloud providers when accessed from different parts of the world. This means that no provider covers all areas in the same way, and highlights another advantage of the cloud-of-clouds: we can adapt our accesses to use the best cloud for a certain location.

The effect of optimizations. An interesting observation of our DEPSKY-A (resp. DEPSKY-CA) read experiments is that in a significant percentage of the reads the cloud that replied metadata faster (resp. the two faster in replying metadata) is not the first to reply the data (resp. the two first in replying the data). More precisely, in 17% of the 60768 DEPSKY-A reads and 32% of the 60444 DEPSKY-CA reads we observed this behavior. A possible explanation for this could be that some clouds are better serving small files (DEPSKY metadata is around 500 bytes) and not so good on serving large files (like the 10MB data unit of some experiments). This means that the read optimizations of Section 3.6 will make the protocol latency worse in these cases. Nonetheless we think this optimization is valuable since the rationale behind it worked for more than 4/5 (DEPSKY-



(a) 100kB DU.

(b) 1MB DU.

(c) 10MB DU.

Fig. 6. $50^{th}/90^{th}$-percentile latency (in seconds) for 100kB, 1MB and 10MB DU read operations with PlanetLab clients located on different parts of the globe. The bar names are S3 for Amazon S3, WA for Windows Azure, NX for Nirvanix, RS for Rackspace, A for DEPSKY-A and CA for DEPSKY-CA. DEPSKY-CA and DEPSKY-A are configured with $n = 4$ and $f = 1$.

A) and 2/3 (DEPSKY-CA) of the reads in our experiments, and its use can decrease the monetary costs of executing a read by a quarter and half of the cost of the non-optimized protocol, respectively.

Table V shows, for each cloud (DEPSKY-A) or pair of clouds (DEPSKY-CA), the percentage of read operations that fetched data files from these clouds (i.e., these clouds answered first) for different client locations.

The first four lines of the table show that Rackspace was the cloud that provided the data file faster for most DEPSKY-A clients, while Amazon S3 provided the data more frequently for European clients. Interestingly, although these two clouds are consistently among the most used in operations coming from different parts of the world, it is difficult to decide between Windows Azure and Nirvanix to compose the preferred quorum to be used. Nirvanix showed to be fast for Asian clients (e.g., 45% of reads in Japan), while Windows Azure provided excellent performance for the UK (e.g., 40% of reads fetched data from it). This tie can be broken considering the expected client location, the performance of writes and the economical costs.

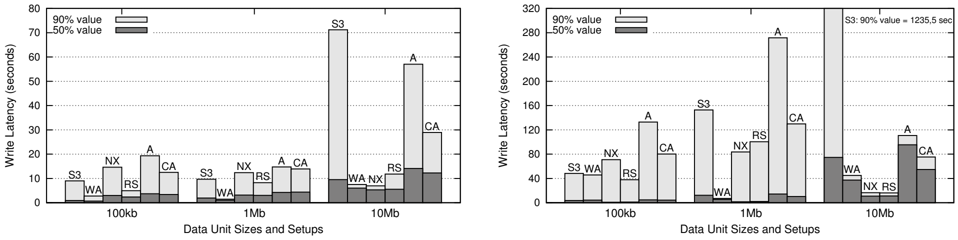
Considering DEPSKY-CA, where two data files are required to rebuild the original data, one can see that there are three possible preferred quorums for different locations: S3-RS-NX (Brazil, US-PA, New Zealand, Japan and Spain), NX-RS-WA (US-CA and China) and S3-WA-RS (UK). Again,

Table V. Percentage of reads in which the required data blocks were fetched from a specific cloud (or pair of clouds) for different locations. The clouds names are S3 for Amazon S3, WA for Windows Azure, NX for Nirvanix and RS for Rackspace. Results for single clouds stand for DEPSKY-A reads while results for a pair of clouds correspond to the 2 blocks read in DEPSKY-CA to rebuild the data.

Cloud(s)	Brazil	US-PA	US-CA	New Z.	Japan	China	Spain	UK
S3	4	3	0	1	0	1	65	59
NX	0	2	0	14	45	2	2	0
RS	94	94	99	84	55	97	31	0
WA	1	1	0	0	-	0	2	40
S3-RS	53	61	2	3	1	3	67	2
S3-NX	0	1	0	0	0	1	3	0
S3-WA	0	1	-	0	-	0	2	81
NX-WA	0	1	0	0	0	1	1	6
NX-RS	30	20	87	97	99	81	15	0
RS-WA	17	16	11	0	0	14	12	10

the choice of the quorum used initially needs to be based on the other factors already mentioned. If one considers only the cost factor, the choice would be S3-RS-WA for both DEPSKY-A and DEPSKY-CA, since Windows Azure is much less expensive than Nirvanix (see Figure 5(b) in Section 8.1). On the other hand, as will be seen in the following, the perceived availability of Windows Azure was worse than Nirvanix in our experiments.

Writes. We modified our logger application to execute writes instead of reads and deployed it on the same machines we executed the reads. We ran it for two days in October 2010 and collected the logs, with at least 500 measurements for each location and data size. These experiments were executed without the (monetary) read optimization described in Section 3.6. For the sake of brevity, we do not present all these results, but illustrate the costs of write operations for different data sizes and locations discussing only the observed results for UK and US-CA clients. The other locations present similar trends. These experiments were executed without the preferred quorum optimization described in Section 3.6. The 50% and 90% percentile of the latencies observed are presented in Figure 7.



(a) UK.

(b) US-CA.

Fig. 7. $50^{th}/90^{th}$-percentile latency (in seconds) for 100kB, 1MB and 10MB DU write operation for a PlanetLab client at the UK (a) and US-CA (b). The bar names are the same as in Figure 6. DEPSKY-A and DEPSKY-CA are configured with $n = 4$ and $f = 1$.

The latencies in the figure consider the time of writing the data on all four clouds (file sent to 4 clouds, wait for only 3 confirmations) and the time of writing the new metadata. As can be observed in the figure, the latency of a write is of the same order of magnitude of a read of a DU of the same size (this was observed on all locations). It is interesting to observe that, while DEPSKY's read latency is close to the cloud with best latency, the write latency is close to the worst cloud.

This comes from the fact that in a write DEPSKY needs to upload data blocks to all clouds, which consumes more bandwidth at the client side and requires replies from at least three clouds.

The figure also illustrates the big differences between the performance of the system depending on the client location. This difference is specially relevant when looking at the 90% values reported.

Secret sharing overhead. As discussed in Section 3.5, if a key distribution mechanism is available, secret sharing could be removed from DEPSKY-CA. However, the effect of this on read and write latencies would be negligible since *share* and *combine* (lines 9 and 34 of Algorithm 2) account for less than 3 and 0.5 ms, respectively. It means that secret sharing is responsible for less than 0.1% of the protocol's latency in the worst case[3].

Throughput. Table VI shows the throughput in the experiments for two locations: UK and US-CA. The values are of the throughput observed by a single client, not by multiple clients as done in some throughput experiments. The table shows read and write throughput for both DEPSKY-A and DEPSKY-CA, together with the values observed from Amazon S3, just to give a baseline. The results from other locations and clouds follow the same trends discussed here.

Table VI. Throughput observed in kB/s on all reads and writes executed for the case of 4 clouds ($f = 1$).

Operation	DU Size	UK			US-CA		
		DEPSKY-A	DEPSKY-CA	Amazon S3	DEPSKY-A	DEPSKY-CA	Amazon S3
Read	100kB	189	135	59.3	129	64.9	31.5
	1MB	808	568	321	544	306	104
	10MB	1479	756	559	780	320	147
Write	100kB	3.53	4.26	5.43	2.91	3.55	5.06
	1MB	14.9	26.2	53.1	13.6	19.9	25.5
	10MB	64.9	107	84.1	96.6	108	34.4

By the table it is possible to observe that the read throughput decreases from DEPSKY-A to DEPSKY-CA and then to Amazon S3, at the same time that write throughput increases for the same sequence. The higher read throughput of DEPSKY when compared with Amazon S3 is due to the fact that it fetches the data from all clouds at the same time, trying to obtain the data from the fastest cloud available. The price to pay for this benefit is the lower write throughput since data should be written at least on a quorum of clouds in order to complete a write. This trade off appears to be a good compromise since reads tend to dominate most workloads of storage systems.

The table also shows that increasing the size of the data unit improves throughput. Increasing the data unit size from 100kB to 1MB improves the throughput by an average factor of 5 in both reads and writes. By the other hand, increasing the size from 1MB to 10MB shows less benefits: read throughput is increased only by an average factor of 1.5 while write throughput increases by an average factor of 3.3. These results show that cloud storage services should be used for storing large chunks of data. However, increasing the size of these chunks brings less benefit after a certain size (1MB).

Notice that the observed throughputs are at least an order of magnitude lower than the throughput of disk access or replicated storage in a LAN [Hendricks et al. 2007], but the elasticity of the cloud allows the throughput to grow indefinitely with the number of clients accessing the system (according to the cloud providers). This is actually the main reason that lead us to not trying to measure the peak throughput of services built on top of clouds. Another reason is that the Internet bandwidth would probably be the bottleneck of the throughput, not the clouds.

Faults and availability. During our experiments we observed a significant number of read operations on individual clouds that could not be completed due to some error. Table VII presents the *perceived availability* of all setups calculated as $\frac{reads\_completed}{reads\_tried}$ from different locations.

[3] For a more comprehensive discussion about the overhead imposed by Java secret sharing see [Bessani et al. 2008].

Table VII. The perceived availability of all setups evaluated from different points of the Internet. The values were calculated as $\frac{reads\_completed}{reads\_tried}$.

Location	Reads Tried	DepSky-A	DepSky-CA	Amazon S3	Rackspace	Azure	Nirvanix
Brazil	8428	1.0000	0.9998	1.0000	0.9997	0.9793	0.9986
US-PA	5113	1.0000	1.0000	0.9998	1.0000	1.0000	0.9880
US-CA	8084	1.0000	1.0000	0.9998	1.0000	1.0000	0.9996
New Zealand	8545	1.0000	1.0000	0.9998	1.0000	0.9542	0.9996
Japan	8392	1.0000	1.0000	0.9997	0.9998	0.9996	0.9997
China	8594	1.0000	1.0000	0.9997	1.0000	0.9994	1.0000
Spain	6550	1.0000	1.0000	1.0000	1.0000	0.9796	0.9995
UK	7069	1.0000	1.0000	0.9998	1.0000	1.0000	1.0000

The first thing that can be observed from the table is that the number of measurements taken from each location is not the same. This happens due to the natural unreliability of PlanetLab nodes, that crash and restart with some regularity.

There are two key observations that can be taken from Table VII. First, DepSky-A and DepSky-CA are the two single setups that presented an availability of 1.0000 in almost all locations[4]. Second, despite the fact that most cloud providers advertise providing 5 or 6 nines of availability, the perceived availability in our experiments was lower. The main problem is that outsourcing storage makes a company not only dependent on the provider's availability, but also on the network availability, which some studies show to have no more than two nines of availability [Dahlin et al. 2003]. This is a fact that companies moving critical applications to the cloud have to be fully aware of.

9. RELATED WORK

Byzantine quorum systems. DepSky provides a single-writer multi-reader read/write register abstraction built on a set of untrusted storage clouds that can fail in an arbitrary way. This type of abstraction supports an updatable data model, requiring protocols that can handle multiple versions of stored data. This is substantially different from providing write-once, read-maybe archival storage such as the one described in [Storer et al. 2007].

There are many protocols for Byzantine quorums systems for register implementation (e.g., [Goodson et al. 2004; Hendricks et al. 2007; Malkhi and Reiter 1998a; Martin et al. 2002]), however, few of them address the model in which servers are passive entities that do not run protocol code [Abraham et al. 2006; Attiya and Bar-Or 2003; Jayanti et al. 1998]. DepSky differentiates from them in the following aspects: (1.) it decouples the write of timestamp and verification data from the write of the new value; (2.) it has optimal resiliency ($3f+1$ servers [Martin et al. 2002]) and employs read and write protocols requiring two communication round-trips independently of the existence of contention, faults and weakly consistent clouds; finally, (3.) it is the first single-writer multi-reader register implementation supporting efficient encoding and confidentiality. Regarding (2.), our protocols are similar to others for fail-prone shared memory (or "disk quorums"), where servers are passive disks that may crash or corrupt stored data. In particular, Byzantine disk Paxos [Abraham et al. 2006] also presents a single-writer multi-reader regular register construction that requires two communication round-trips both for reading and writing in absence of contention. However, there is a fundamental difference between this construction and DepSky: it provides a weak liveness condition for the read protocol (termination only when there is a finite number of contending writes) while our protocol satisfies wait-freedom. An important consequence of this limitation is that reads may require several communication steps when contending writes are being executed. This same limitation appears on [Attiya and Bar-Or 2003] that, additionally, does not tolerate writer faults. Regarding point (3.), it is worth to notice that several Byzantine storage protocols support efficient storage using erasure codes [Cachin and Tessaro 2006; Goodson et al. 2004; Hendricks et al. 2007], but none of them mention the use of secret sharing or the provision of confidentiality.

[4] This is somewhat surprising since we were expecting to have at least some faults on the client network that would disallow it to access any cloud.

However, it is not clear if information-efficient secret sharing [Krawczyk 1993] or some variant of this technique could substitute the erasure codes employed on these protocols.

Perhaps the closest work to DEPSKY is the recent register emulation protocol suite presented in [Basescu et al. 2012]. This work uses a set of fail-prone key-value stores supporting put, get, delete and list operations (which are equivalent to the untrusted passive storage clouds we consider) in the crash fault model to implement multi-writer multi-reader regular and atomic storage. Albeit with similar objectives, there are many differences between this work and DEPSKY. Although it goes one step beyond DEPSKY to provide genuine multi-writer storage, it assumes a simpler fault model (crash), requires strong consistency from the key-value stores (linearizability) and does not address privacy and cost-efficiency (does not integrate the register emulation with secret sharing or erasure codes). Moreover, this work is also more theoretical-oriented and does not provide a rich experimental evaluation as we do in DEPSKY.

Cloud storage availability. Cloud storage is a hot topic with several papers appearing recently. However, most of these papers deal with the intricacies of implementing a storage infrastructure inside a cloud provider (e.g., [McCullough et al. 2010]). Our work is closer to others that explore the use of existing cloud storage services to implement enriched storage applications. There are papers showing how to efficiently use storage clouds for file system backup [Vrable et al. 2009], implement a database [Brantner et al. 2008], implement a log-based file system [Vrable et al. 2012] or add provenance to the stored data [Muniswamy-Reddy et al. 2010]. However none of these works provide guarantees like confidentiality and availability or considers a cloud-of-clouds.

Some works on this trend deal with the high-availability of stored data through the replication of this data on several cloud providers, and thus are closely related to DEPSKY. The SafeStore system [Kotla et al. 2007] provides an accountability layer for using a set of untrusted third-party storage systems in an efficient way. There are at least two features that make SafeStore very different from DEPSKY. First, it requires specific server-code on the storage cloud provider (both in the service interface and in the internal storage nodes). Second, SafeStore does not support data sharing among clients (called SafeStore local servers) accessing the same storage services. The HAIL (High-Availability Integrity Layer) protocol set [Bowers et al. 2009] combines cryptographic protocols for proof of recoveries with erasure codes to provide a software layer to protect the integrity and availability of the stored data, even if the individual clouds are compromised by a malicious and mobile adversary. HAIL has at least three limitations when compared with DEPSKY: it only deals with static data (i.e., it is not possible to manage multiple versions of data), it requires that the servers run some code, and does not provide confidentiality guarantees for the stored data.

HAIL and SafeStore are examples of replicated storage systems that use auditing protocols, usually based on algebraic signatures and erasure codes [Schwarz and Miller 2006], to verify that the storage server (or cloud) still has the data and that it is not corrupted. DEPSKY could be extended to support such remote auditing operations if the clouds could run a small portion of code to participate in the challenge-response protocol involved in such auditing.

The RACS (Redundant Array of Cloud Storage) system [Abu-Libdeh et al. 2010] employs RAID-like techniques (mainly erasure codes) [Patterson et al. 1988] to implement high-available and storage-efficient data replication on diverse clouds. Differently from DEPSKY, RACS does not try to solve security problems of cloud storage, but instead deals with "economic failures" and vendor lock-in. In consequence, the system does not provide any mechanism to detect and recover from data corruption or confidentiality violations. Moreover, it does not provide updates of the stored data. Finally, it is worth to mention that none of these works on cloud replication present an experimental evaluation with diverse clouds as it is presented in this paper.

Cloud security. There are several works about obtaining trustworthiness from untrusted clouds. Depot improves the resilience of cloud storage making similar assumptions to DEPSKY, namely that storage clouds are fault-prone black boxes [Mahajan et al. 2011]. However, it uses a single cloud, so it provides a solution that is cheaper but does not tolerate total data losses and the availability is constrained by the availability of the cloud on top of which it is implemented. Works like SPORC

[Feldman et al. 2010] and Venus [Shraer et al. 2010] make similar assumptions to implement services on top of untrusted clouds. All these works consider a single cloud (not a cloud-of-clouds), require a cloud with the ability to run code, and have limited support for cloud unavailability, which makes them different from DEPSKY.

10. CONCLUSION

This paper presents the design, implementation and evaluation of DEPSKY, a storage service that improves the availability and confidentiality provided by commercial storage clouds. The system achieves these objectives by building a cloud-of-clouds on top of a set of storage clouds, combining Byzantine quorum system protocols, cryptographic secret sharing, erasure codes and the diversity provided by the use of several clouds. Moreover, the notion of consistency proportionality introduced by DEPSKY allows the system to provide the same level of consistency of the underlying clouds it uses for storage.

We believe DEPSKY protocols are in an unexplored region of the quorum systems design space and can enable applications sharing critical data (e.g., financial, medical) to benefit from storage clouds. Moreover, the few and weak assumptions required by the protocols allow them to be used to replicate data efficiently not only on cloud storage services, but with any storage service available (e.g., NAS disks, NFS servers, FTP servers, key-value databases).

The paper also presents an extensive evaluation of the system. The key conclusion is that it provides confidentiality and improved availability with an added cost as low as 23% of the cost of storing data on a single cloud for a practical scenario, which seems to be a good compromise for critical applications.

ACKNOWLEDGMENTS

We warmly thank Bernhard Kauer and the TClouds FP-7 project partners for the many insightful discussions, suggestions and encouragement that lead us to prepare this extended and improved version of the original DEPSKY paper. We also thank the EuroSys'11 and ACM TOS reviewers for their comments on earlier versions of this paper.

A. AUXILIARY FUNCTIONS

Algorithm 4 presents the two auxiliary functions described in Table I and used in Algorithms 1 and 2. These two functions are similar and equally simple: the process just accesses all the n clouds in parallel to get or put data and waits for replies from a quorum of clouds, canceling non-answered RPCs.

B. STORAGE PROTOCOLS CORRECTNESS

This section presents correctness proofs of the DEPSKY-A and DEPSKY-CA protocols. The first lemma states that the auxiliary functions presented in Appendix A are wait-free.

LEMMA B.1. *A correct process will not block executing writeQuorum or queryMetadata.*

PROOF. Both operations require $n-f$ clouds to answer the put and get requests. For *writeQuorum*, these replies are just *acks* and they will always be received since at most f clouds can be faulty. For the *queryMetadata*, the operation is finished only if one metadata file is available. Since we are considering only non-malicious writers, a metadata written in a cloud is always valid and thus correctly signed using $K_{r_w}^{du}$. It means that a valid metadata file will be read from at least $n-f$ clouds and the process will choose one of these files and finish the algorithm. □

The next two lemmas state that if a correctly signed metadata is obtained from the cloud providers (using *queryMetadata*) the corresponding data can also be retrieved and that the metadata stored on DEPSKY-A and DEPSKY-CA satisfy the properties of a regular register [Lamport 1986] (if the clouds provide this consistency semantics).

ALGORITHM 4: DEPSKY-A and DEPSKY-CA auxiliary functions.

```
function $queryMetadata(du)$
begin
    $m[0..n-1] \longleftarrow \bot$
    parallel for $0 \leq i \leq n-1$ do
        $tmp_i \longleftarrow cloud_i.get(du,$ "metadata" $)$
        if $verify(tmp_i, K^{du}_{u_w})$ then $m[i] \longleftarrow tmp_i$
    wait until $|\{i : m[i] \neq \bot\}| \geq n-f$
    for $0 \leq i \leq n-1$ do $cloud_i.cancel\_pending()$
    return $m$

procedure $writeQuorum(du, name, value)$
begin
    $ok[0..n-1] \longleftarrow false$
    parallel for $0 \leq i \leq n-1$ do
        $ok[i] \longleftarrow cloud_i.put(du, name[i], value[i])$
    wait until $|\{i : ok[i] = true\}| \geq n-f$
    for $0 \leq i \leq n-1$ do $cloud_i.cancel\_pending()$
```

LEMMA B.2. *The value associated with the metadata with greatest version number returned by queryMetadata, from now on called* outstanding metadata*, is available on at least $f+1$ non-faulty clouds.*

PROOF. Recall that only valid metadata files will be returned by *queryMetadata*. These metadata will be written only by a non-malicious writer that follows the DepSkyAWrite (resp. DepSky-CAWrite) protocol. In this protocol, the data value is written on a quorum of clouds on line 8 (resp. line 14) of Algorithm 1 (resp. Algorithm 2), and then the metadata is generated and written on a quorum of clouds on lines 9-12 (resp. lines 15-18). Consequently, a metadata is only put on a cloud if its associated value was already put on a quorum of clouds. It implies that if a metadata is read, its associated value was already written on $n-f$ clouds, from which at least $n-f-f \geq f+1$ are correct. □

LEMMA B.3. *The outstanding metadata obtained on a DepSkyARead (resp. DepSkyCARead) concurrent with zero or more DepSkyAWrites (resp. DepSkyCAWrites) is the metadata written on the last complete write or being written by one of the concurrent writes.*

PROOF. Assuming that a client reads an outstanding metadata m, we have to show that m was written on the last complete write or is being written concurrently with the read. This proof can easily be obtained by contradiction. Suppose m was written before the start of the last complete write preceding the read and that it was the metadata with greatest version number returned from *queryMetadata*. This is clearly impossible because m was overwritten by the last complete write (which has a greater version number) and thus will never be selected as the outstanding metadata. It means that m can only correspond to the last complete write or to a write being executed concurrently with the read. □

With the previous lemmas we can prove the wait-freedom of the DEPSKY-A and DEPSKY-CA registers, showing that their operations will never block.

THEOREM B.4. *All* DEPSKY *read and write operations are wait-free operations.*

PROOF. Both Algorithms 1 and 2 use functions *queryMetadata* and *writeQuorum*. As shown in Lemma B.1, these operations can not block. Besides that, read operations make processes wait for the value associated with the outstanding metadata. Lemma B.2 states that there are at least $f+1$ correct clouds with this data, and thus at least one of them will answer the RPC of lines 21 and 27 of Algorithms 1 and 2, respectively, with values that match the digest contained on the metadata (or the different block digests in the case of DEPSKY-CA) and make $d[i] \neq \perp$, releasing itself from the barrier and completing the algorithm. □

The next two theorems show that DEPSKY-A and DEPSKY-CA implement single-writer multi-reader regular registers.

THEOREM B.5. *A client reading a* DEPSKY-A *register in parallel with zero or more writes (by the same writer) will read the last complete write or one of the values being written.*

PROOF. Lemma B.3 states that the outstanding metadata obtained on lines 16-17 of Algorithm 1 corresponds to the last write executed or one of the writes being executed. Lemma B.2 states that the value associated with this metadata is available from at least $f+1$ correct clouds, thus it can be obtained by the client on lines 20-24: just a single valid reply will suffice for releasing the barrier of line 24 and return the value. □

THEOREM B.6. *A client reading a* DEPSKY-CA *register in parallel with zero or more writes (by the same writer) will read the last complete write or one of the values being written.*

PROOF. This proof is similar to the one for DEPSKY-A. Lemma B.3 states that the outstanding metadata obtained on lines 22-23 of Algorithm 2 corresponds to the last write executed or one of the writes being executed concurrently. Lemma B.2 states that the values associated with this metadata are stored on at least $f+1$ non-faulty clouds, thus a reader can obtain them through the execution of lines 26-30: all non-faulty clouds will return their values corresponding to the outstanding metadata allowing the reader to decode the encrypted value, combine the key shares and decrypt the read data (lines 33-35), inverting the processing done by the writer on DepSkyCAWrite (lines 7-10). □

C. LOCK PROTOCOL CORRECTNESS

This section presents correctness proofs for the data unit locking protocol. Recall from Section 4.1 that this protocol requires two extra assumptions: (1) contending writers have their clocks synchronized with precision $\Delta/2$ and (2) the storage clouds provides at least read-after-write consistency.

Before the main proofs, we need to present a basic lemma that shows that a lock file created in a quorum of clouds is read in a later listing of files from a quorum of clouds.

LEMMA C.1. *An object o created with the operation writeQuorum(du,o,v) and not removed will appear in at least one result of later list(du) operations executed on a quorum of clouds.*

PROOF. The *writeQuorum*(du,o,v) operation is only completed when the object is created/written in a quorum of at least $n-f$ clouds (line 15 of Algorithm 4). If a client tries to list the objects of du on a quorum of clouds, at least one of the $n-f$ clouds will provide it since there is at least one correct cloud between any two quoruns $((n-f)+(n-f)-n>f)$. □

In order to prove the mutual exclusion on lock possession we need to precisely define what it means for a process to hold the lock for a given data unit.

Definition C.2. A correct client c is said to *hold the write lock for a du at a given time t* if an object du-lock-c-T containing $sign(\text{du-lock-c-T}, K_c)$ with $T+\Delta<t$ appears in at least one $list(du)$ result when this operation is executed in a quorum of clouds.

With this definition, we can prove the safety and liveness properties of Algorithm 3.

THEOREM C.3 (MUTUAL EXCLUSION). *At any given time t, there is at most one correct client that holds the lock for a data unit du.*

PROOF. Assume this is false: there is a time t in which two correct clients c_1 and c_2 hold the lock for du. We will prove that this assumption leads to a contradiction.

If both c_1 and c_2 hold the write lock for du we have that both du-lock-c_1-T_1 and du-lock-c_2-T_2 with $T_1+\Delta<t$ and $T_2+\Delta<t$ are returned in $list(du)$ operations from a quorum of clouds.

Algorithm 3 and Lemma C.1 states that it can only happens if both c_1 and c_2 wrote valid lock files (line 12) and did not remove them (line 19). In order for this to happen, both c_1 and c_2 must see only their lock files in their second $list(du)$ on the clouds (lines 14-16).

Two situations may arise when c_1 and c_2 acquire write locks for du: either c_1 (resp. c_2) writes its lock file before c_2 (resp. c_1) lists the lock files the second time (i.e., either w_1 precedes r_2 or w_2 precedes r_1) or one's lock file is being written while the other is listing lock files for the second time (i.e., either w_1 is executed concurrently with r_2 or w_2 is executed concurrently with r_1).

In the first situation, when c_2 (resp. c_1) lists available locks, it will see both lock files and thus remove du-lock-c_2-T_2 (resp. du-lock-c_1-T_1), releasing the lock (lines 14-20).

The second situation is more complicated because now we have to analyze the start and finish of each phase of the algorithm. Consider the case in which c_1 finishes writing its lock file (line 12) after c_2 executes the second list (lines 14-15). Clearly, in this case c_2 may or may not see du-lock-c_1-T_1 in line 18. However, we can say that the second list of c_1 will see du-lock-c_2-T_2 since it is executed after c_1 lock file is written, which happens, only after c_2 start its second list, and consequently after its lock file is written. It means that the condition of line 18 will be true for c_1, and it will remove du-lock-c_1-T_1, releasing its lock. The symmetric case (c_2 finishes writing its lock after c_1 executes the second list) also holds.

In both situations we have a contradiction, i.e., it is impossible to have an execution and time in which two correct clients hold the lock for du. □

THEOREM C.4 (OBSTRUCTION-FREEDOM). *If a correct client tries to obtain the lock for a data unit du without contention it will succeed.*

PROOF. When there is no other valid lock in the cloud (i.e., the condition of line 10 holds), c will write du-lock-c-T on a quorum of clouds. This lock file will be the only valid lock file read on the second list (the condition of line 18 will not hold) since (1.) no other valid lock file is available on the clouds, (2.) no other client is trying to acquire the lock, and (3.) Lemma C.1 states that if the lock file was written it will be read. After this, c acquire the lock and return it. □

D. CONSISTENCY PROPORTIONALITY

In this section we prove the consistency proportionality of the DEPSKY-A and DEPSKY-CA protocols considering some popular consistency models [Lamport 1986; Terry et al. 1994; Vogels 2009].

In the following theorem we designate by the *weakest cloud* the correct cloud that provides less guarantees in terms of consistency. In homogeneous environments, all clouds will provide the same consistency, but in heterogeneous environments other clouds will provide at least the guarantees of the weakest cloud.

THEOREM D.1. *If the weakest cloud used in a* DEPSKY-CA *setup satisfies a consistency model* $\mathscr{C}$*, the data unit provided by* DEPSKY-CA *also satisfies* $\mathscr{C}$ *for any* $\mathscr{C} \in \{$eventual, read-your-writes, monotonic reads, writes-follow-reads, monotonic writes, read-after-write$\}$.

PROOF (SKETCH). Notice that cloud consistency issues only affect metadata readings since in the DEPSKY-CA (Algorithm 2), after the *max_id* variable is defined (line 23), the clients keep reading the clouds until the data value is read (lines 24-32). So, even with eventual consistency (the weakest guarantee we consider), if the metadata file pointing to the last version is read, the data will eventually be read.

Let Q_w be the quorum of clouds in which the metadata of the last executed write w was written and let Q_r be the quorum of clouds where *queryMetadata* obtained an array of $n-f$ metadata files on a posterior read r. Let $cloud \in Q_w \cap Q_r$ be the *weakest cloud* among the available clouds. For each of the considered consistency models, we will prove that if *cloud* provides this consistency, the register implemented by DEPSKY-CA provides the same consistency.

For *eventual consistency* [Vogels 2009], if the outstanding metadata file was written on *cloud* and no other write operation is executed, it will be eventually available for reading in this cloud, and then its associated data will be fetched from the clouds. As a consequence, the data described in the metadata file will be read eventually, satisfying this model.

For *read-your-writes* consistency, if both w and r are executed by c, the fact that *cloud* provides this consistency means that at least this cloud will return the metadata written in w during r execution. Consequently, the result of the read will be the value written in w, satisfying this model.

For *monotonic reads* consistency, assume c executed r and also another posterior read r'. Let $Q_{r'}$ be the quorum accessed when reading metadata file on r'. Let $cloud' \in Q_r \cap Q_{r'}$ be a *correct* cloud providing *monotonic reads* consistency. We have to prove that r' will return the same data of r or a value written in a posterior write. Since $cloud'$ satisfy monotonic reads, the metadata file read in r' will be either the one read in r or another one written by a posterior write. In any of the two cases, the corresponding value returned will satisfy the monotonic reads consistency.

For *writes-follow-reads* consistency, the result is trivial since, as long as we have no contending writers, the metadata files are written with increasing version numbers. Since the clouds provide at least this consistency, it is impossible to observe (and to propagate) writes in a different order they were executed, i.e., to observe them in an order different from the version numbers of their metadata.

The same arguments holds for *monotonic writes*: since the clouds provide at least this consistency, it is impossible to observe the writes in a different order they were executed.

Finally, for *read-after-write* consistency[5], the safety properties proved in previous section (see Theorem B.6) can be easily generalized for any read-after-write model. If the outstanding metadata file was written on a *cloud* satisfying this consistency model during write w and no other write operation is executed, any read succeeding w will see this file, and its associated data will be fetched from the cloud. □

Since the critical step of Theorem D.1's proof uses the intersection between metadata's reads and writes, the following corollary states that the result just proved for DEPSKY-CA is also valid for DEPSKY-A. The key reason is that both protocols read and validate metadata files in the same way, as can be seen in lines 16-17 of Algorithm 1 and 22-23 of Algorithm 2).

COROLLARY D.2. *If the weakest cloud used in a* DEPSKY-A *setup satisfies a consistency model* $\mathscr{C}$*, the data unit provided by* DEPSKY-A *also satisfies* $\mathscr{C}$ *for any* $\mathscr{C} \in \{$eventual, read-your-writes, monotonic reads, writes-follow-reads, monotonic writes, read-after-write$\}$.

References

ABRAHAM, I., CHOCKLER, G., KEIDAR, I., AND MALKHI, D. 2006. Byzantine disk Paxos: optimal resilience with Byzantine shared memory. *Distributed Computing 18,* 5, 387–408.

ABU-LIBDEH, H., PRINCEHOUSE, L., AND WEATHERSPOON, H. 2010. RACS: A case for cloud storage diversity. *Proc. of the 1st ACM Symposium on Cloud Computing*, 229–240.

ALCHIERI, E. A. P., BESSANI, A. N., AND FRAGA, J. D. S. 2008. A dependable infrastructure for cooperative web services coordination. In *Proc. of the 2008 IEEE Int'l Conference on Web Services*. 21–28.

AMAZON WEB SERVICES. 2011. Amazon simple storage service faqs. `http://aws.amazon.com/s3/faqs/`.

ATTIYA, H. AND BAR-OR, A. 2003. Sharing memory with semi-Byzantine clients and faulty storage servers. In *Proc. of the 22rd IEEE Symposium on Reliable Distributed Systems - SRDS 2003*. 174–183.

BASESCU, C., CACHIN, C., EYAL, I., HAAS, R., SORNIOTTI, A., VUKOLIC, M., AND ZACHEVSKY, I. 2012. Robust data sharing with key-value stores. In *Proc. of the 42nd Int'l Conference on Dependable Systems and Networks - DSN 2012*.

BESSANI, A. N., ALCHIERI, E. P., CORREIA, M., AND FRAGA, J. S. 2008. DepSpace: a Byzantine fault-tolerant coordination service. In *Proc. of the 3rd ACM European Systems Conference – EuroSys'08*. 163–176.

BOWERS, K. D., JUELS, A., AND OPREA, A. 2009. HAIL: a high-availability and integrity layer for cloud storage. In *Proc. of the 16th ACM Conference on Computer and Communications Security - CCS'09*. 187–198.

[5] Any of the consistency models introduced in [Lamport 1986] satisfies *read-after-write* since the reads return the value written in the last complete operation in absence of contention.

BRANTNER, M., FLORESCU, D., GRAF, D., KOSSMANN, D., AND KRASKA, T. 2008. Building a database on S3. In *Proc. of the 2008 ACM SIGMOD Int'l Conference on Management of Data*. 251–264.

CACHIN, C. AND TESSARO, S. 2006. Optimal resilience for erasure-coded Byzantine distributed storage. In *Proc. of the Int. Conference on Dependable Systems and Networks - DSN 2006*. 115–124.

CHOCKLER, G., GUERRAOUI, R., KEIDAR, I., AND VUKOLIĆ, M. 2009. Reliable distributed storage. *IEEE Computer 42,* 4, 60–67.

CHOCKLER, G. AND MALKHI, D. 2002. Active disk Paxos with infinitely many processes. In *Proc. of the 21st Symposium on Principles of Distributed Computing – PODC'02*. 78–87.

DAHLIN, M., CHANDRA, B., GAO, L., AND NAYATE, A. 2003. End-to-end WAN service availability. *ACM/IEEE Transactions on Networking 11,* 2.

DEKKER, M. A. C. 2012. Critical Cloud Computing: A CIIP perspective on cloud computing services (v1.0). Tech. rep., European Network and Information Security Agency (ENISA). Dec.

FELDMAN, A. J., ZELLER, W. P., FREEDMAN, M. J., AND FELTEN, E. W. 2010. SPORC: Group collaboration using untrusted cloud resources. In *Proc. of the 9th USENIX Symposium on Operating Systems Design and Implementation – OSDI'10*. 337–350.

GAFNI, E. AND LAMPORT, L. 2003. Disk Paxos. *Distributed Computing 16,* 1, 1–20.

GHEMAWAT, S., GOBIOFF, H., AND LEUNG, S.-T. 2003. The Google file system. In *Proc. of the 19th ACM Symposium on Operating Systems Principles – SOSP'03*. 29–43.

GIBSON, G., NAGLE, D., AMIRI, K., BUTLER, J., CHANG, F., GOBIOFF, H., HARDIN, C., RIEDEL, E., ROCHBERG, D., AND ZELENKA, J. 1998. A cost-effective, high-bandwidth storage architecture. In *Proc. of the 8th Int. Conference on Architectural Support for Programming Languages and Operating Systems - ASPLOS'98*. 92–103.

GOODSON, G., WYLIE, J., GANGER, G., AND REITER, M. 2004. Efficient Byzantine-tolerant erasure-coded storage. In *Proc. of the Int. Conference on Dependable Systems and Networks - DSN'04*. 135–144.

GREER, M. 2010. Survivability and information assurance in the cloud. In *Proc. of the 4th Workshop on Recent Advances in Intrusion-Tolerant Systems – WRAITS'10*.

HAMILTON, J. 2007. On designing and deploying Internet-scale services. In *Proc. of the 21st Large Installation System Administration Conference – LISA'07*. 231–242.

HANLEY, M., DEAN, T., SCHROEDER, W., HOUY, M., TRZECIAK, R. F., AND MONTELIBANO, J. 2011. An analysis of technical observations in insider theft of intellectual property cases. Technical Note CMU/SEI-2011-TN-006, Carnegie Mellon Software Engineering Institute. Feb.

HENDRICKS, J., GANGER, G., AND REITER, M. 2007. Low-overhead byzantine fault-tolerant storage. In *Proc. of the 21st ACM Symposium on Operating Systems Principles – SOSP'07*. 73–86.

HENRY, A. 2009. Cloud storage FUD (failure, uncertainty, and durability). Keynote Address at the 7th USENIX Conference on File and Storage Technologies.

HERLIHY, M., LUCANGCO, V., AND MOIR, M. 2003. Obstruction-free syncronization: double-ended queues as an example. In *Proc. of the 23th IEEE Int. Conference on Distributed Computing Systems - ICDCS 2003*. 522–529.

HERLIHY, M. AND WING, J. M. 1990. Linearizability: A correctness condition for concurrent objects. *ACM Transactions on Programing Languages and Systems 12,* 3, 463–492.

HUNT, P., KONAR, M., JUNQUEIRA, F., AND REED, B. 2010. Zookeeper: Wait-free coordination for Internet-scale services. In *Proc. of the USENIX Annual Technical Conference – USENIX ATC'10*. 145–158.

JAYANTI, P., CHANDRA, T. D., AND TOUEG, S. 1998. Fault-tolerant wait-free shared objects. *Journal of the ACM 45,* 3, 451–500.

KOTLA, R., ALVISI, L., AND DAHLIN, M. 2007. SafeStore: A durable and practical storage system. In *Proc. of the USENIX Annual Technical Conference - USENIX ATC'07.*

KRAWCZYK, H. 1993. Secret sharing made short. In *Proc. of the 13th Int. Cryptology Conference – CRYPTO'93*. 136–146.

LAMPORT, L. 1986. On interprocess communication (part II). *Distributed Computing 1,* 1, 203–213.

LAMPORT, L., SHOSTAK, R., AND PEASE, M. 1982. The Byzantine generals problem. *ACM Transactions on Programing Languages and Systems 4,* 3, 382–401.

LISKOV, B. AND RODRIGUES, R. 2006. Tolerating Byzantine faulty clients in a quorum system. In *Proc. of the 26th IEEE Int. Conference on Distributed Computing Systems - ICDCS'06.*

MAHAJAN, P., SETTY, S., LEE, S., CLEMENT, A., ALVISI, L., DAHLIN, M., AND WALFISH, M. 2011. Depot: Cloud storage with minimal trust. *ACM Transactions on Computer Systems 29,* 4.

MALKHI, D. AND REITER, M. 1998a. Byzantine quorum systems. *Distributed Computing 11,* 4, 203–213.

MALKHI, D. AND REITER, M. 1998b. Secure and scalable replication in Phalanx. In *Proc. of the 17th IEEE Symposium on Reliable Distributed Systems - SRDS'98*. 51–60.

MARTIN, J.-P., ALVISI, L., AND DAHLIN, M. 2002. Minimal Byzantine storage. In *Proc. of the 16th Int. Symposium on Distributed Computing – DISC 2002*. 311–325.

MAY, M. 2010. Forecast calls for clouds over biological computing. *Nature Medicine 16,* 6.

MCCULLOUGH, J. C., JOHNDUNAGAN, WOLMAN, A., AND SNOEREN, A. C. 2010. Stout: An adaptive interface to scalable cloud storage. In *Proc. of the USENIX Annual Technical Conference – USENIX ATC'10*. 47–60.

METZ, C. 2009. DDoS attack rains down on Amazon cloud. *The Register*. http://www.theregister.co.uk/2009/10/05/amazon_bitbucket_outage/.

MILLS, D. L. 1992. Network time protocol (version 3): Specification, implementation and analysis. IETF RFC 1305.

MUNISWAMY-REDDY, K.-K., MACKO, P., AND SELTZER, M. 2010. Provenance for the cloud. In *Proc. of the 8th USENIX Conference on File and Storage Technologies – FAST'10*. 197–210.

NAONE, E. 2009. Are we safeguarding social data? Technology Review published by MIT Review, http://www.technologyreview.com/blog/editors/22924/.

PATTERSON, D. A., GIBSON, G., AND KATZ, R. H. 1988. A case for redundant arrays of inexpensive disks (raid). In *Proc. of the 1988 ACM SIGMOD Int'l Conference on Management of Data*. 109–116.

PLANK, J. S. 2007. Jerasure: A library in C/C++ facilitating erasure coding for storage applications. Tech. Rep. CS-07-603, University of Tennessee. September.

RABIN, M. 1989. Efficient dispersal of information for security, load balancing, and fault tolerance. *Journal of the ACM 36,* 2, 335–348.

RAPHAEL, J. 2011. The 10 worst cloud outages (and what we can learn from them). Infoworld. Available at http://www.infoworld.com/d/cloud-computing/the-10-worst-cloud-outages-and-what-we-can-learn-them-902.

SARNO, D. 2009. Microsoft says lost sidekick data will be restored to users. *Los Angeles Times*.

SCHOENMAKERS, B. 1999. A simple publicly verifiable secret sharing scheme and its application to electronic voting. In *Proc. of the 19th Int. Cryptology Conference – CRYPTO'99*. 148–164.

SCHWARZ, T. AND MILLER, E. L. 2006. Store, forget, and check: Using algebraic signatures to check remotely administered storage. In *Proc. of 26th IEEE Int'l Conference on Distributed Computing Systems - ICDCS 2006*.

SHAMIR, A. 1979. How to share a secret. *Communications of ACM 22,* 11, 612–613.

SHRAER, A., CACHIN, C., CIDON, A., KEIDAR, I., MICHALEVSKY, Y., AND SHAKET, D. 2010. Venus: Verification for untrusted cloud storage. In *Proc. of the ACM Cloud Computing Security Workshop – CCSW'10*.

STORER, M. W., GREENAN, K. M., MILLER, E. L., AND VORUGANTI, K. 2007. POTSHARDS: Secure long-term storage without encryption. In *Proc. of the USENIX Annual Technical Conference – USENIX ATC'07*. 143–156.

TERRY, D. B., DEMERS, A. J., PETERSEN, K., SPREITZER, M. J., THEIMER, M. M., AND WELCH, B. B. 1994. Session guarantees for weakly consistent replicated data. In *Proc. of the 3rd Int'l Conference on Parallel and Distributed Information Systems*. 140–149.

VOGELS, W. 2009. Eventually consistent. *Communications of the ACM 52,* 1, 40–44.

VRABLE, M., SAVAGE, S., AND VOELKER, G. M. 2009. Cumulus: Filesystem backup to the cloud. *ACM Transactions on Storage 5,* 4, 1–28.

VRABLE, M., SAVAGE, S., AND VOELKER, G. M. 2012. BlueSky: A cloud-backed file system for the enterprise. In *Proc. of the 10th USENIX Conference on File and Storage Technologies – FAST'12*.

VUKOLIC, M. 2010. The Byzantine empire in the intercloud. *ACM SIGACT News 41,* 3, 105–111.

WEIL, S. A., BRANDT, S. A., MILLER, E. L., LONG, D. D. E., AND MALTZAHN, C. 2006. Ceph: A scalable, high-performance distributed file system. In *Proc. of the 7th USENIX Symposium on Operating Systems Design and Implementation – OSDI 2006*. 307–320.

Received xx; revised xx; accepted xx