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

Alysson Bessani Miguel Correia Bruno Quaresma Fernando André Paulo Sousa *

University of Lisbon, Faculty of Sciences, Portugal

{bessani,mpc}@di.fc.ul.pt {bmmrq84,andr3.fm,pjsousa}@gmail.com

Abstract

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, power system historical information 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 paper 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 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

Keywords Cloud computing, Cloud storage, Byzantine quorum systems

1. Introduction

The increasing maturity of cloud computing technology is leading many organizations to migrate their IT infrastructure and/or adapting 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 by invoking operations in several individual clouds. More specifically, DEPSKY addresses four important limitations of cloud computing for data storage in the following way:

Loss of availability: temporary partial unavailability of the Internet is a well-known phenomenon. When data is moved from inside of the company network to an external datacenter, it is inevitable that service unavailability will be experienced. The same problem can be caused 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 by exploiting replication and diversity to store 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 data.

Loss of privacy: the cloud provider has access to both the data stored in the cloud and metadata like access patterns. The provider may be trustworthy, but malicious insiders are a wide-spread security problem. This is an especial concern in applications that involve keeping private data like health records. An obvious solution is the customer encrypting the

* Now at Maxdata Informática, Portugal.

EuroSys'11, April 10–13, 2011, Salzburg, Austria.

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: there is currently some concern that a few cloud computing providers become dominant, the so called vendor lock-in issue [Abu-Libdeh 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 two 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 these 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 code [Cachin 2006, Goodson 2004, Malkhi 1998a;b, Martin 2002], not possible in storage clouds, or are sensible to contention (e.g., [Abraham 2006]), which makes them difficult to use for geographically dispersed systems with high and variable access latencies. DEPSKY overcomes these limitations by not requiring 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.

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 an efficient set of 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. 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, Nirvanix and Rackspace) and PlanetLab to run clients that access the service from several places worldwide.

2. Cloud Storage Applications

Examples of applications that can benefit from DEPSKY are the following:

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. Recently, the US government 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 European Commission is also investing in the area through FP7 projects like TCLOUDS [tcl 2010].

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 payper-use model is specially appealing. An example would be power system operators. Considering only the case of storage, power systems have data 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 has been being deployed in the UK for some years [Ehs 2010]. 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.

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 (Section 3) 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.

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. Systems like databases, file systems, objects stores and key-value databases can use the cloud as storage layer as long as caching and weak consistency models are used to avoid paying the price of cloud access on every operation.

3. The DEPSKY System

This section presents the DEPSKY system. It starts by presenting the system architecture, then defines the data and system models, the two main algorithms (DEPSKY-A and DEPSKY-CA), and a set of auxiliary protocols.

3.1 DEPSKY Architecture

Figure 1 presents the architecture of DEPSKY. 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 1998], similar to what is used by parallel file systems (e.g., [Ghemawat 2003, Weil 2006]), allowing reads and writes in the back-end (in this case, the untrusted clouds).

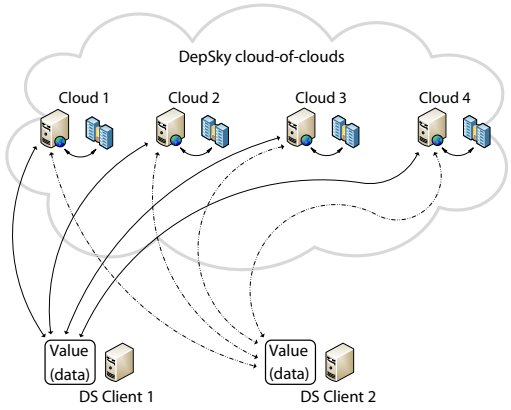


Figure 1. Architecture of DEPSKY (w/ 4 clouds, 2 clients).

3.2 Data Model

The use of diverse clouds requires the DEPSKY library to deal with the heterogeneity of the interfaces of each cloud provider. An aspect that is specially important is the format of the data accepted by each cloud. The data model allow 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 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 *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 informations 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., *value1*, *value2*, 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.).

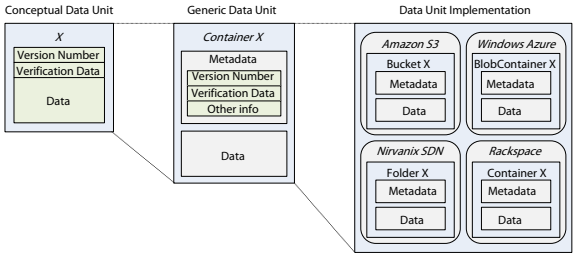


Figure 2. DEPSKY data unit and the 3 abstraction levels.

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.

3.3 System Model

We consider an *asynchronous distributed system* composed by 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 only assumed to fail 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 2006, Liskov 2006]), with active servers verifying the consistency of writer messages, which cannot be implemented on general storage clouds (Section 3.4).

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 system in order to ensure that readers are only allowed to invoke the list and get operations.

Since we do not trust clouds individually, we assume they can fail in a Byzantine way [Lamport 1982]: data stored 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 by 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 2002].

The register abstraction provided by DEPSKY 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]. This semantics is both intuitive and stronger than the eventual consistency of some cloud-based services [Vogels 2009]. Nevertheless, if the semantics provided by the underlying storage clouds is weaker than regular, then DEPSKY's semantics is also weaker (Section 3.10).

Notice that our model hides most of the complexity of the distributed storage system employed by the cloud provider to manage the data storage service since it just assumes that the service is an object storage system prone to Byzantine faults 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 an answer is received or the operation is canceled (possibly by another thread, using a *cancel_pending* special operation to stop resending a request). This means that we have at most once semantics for the operations being invoked. This 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.

3.4 Protocol Design Rationale

Quorum protocols can serve as the backbone of highly available storage systems [Chockler 2009]. There are many protocols for implementing Byzantine fault-tolerant (BFT) storage [Cachin 2006, Goodson 2004, Hendricks 2007, Liskov 2006, Malkhi 1998a;b, Martin 2002], but most of them require that the servers execute some code, a functionality not available on storage clouds. This leads to a key difference between the DEPSKY protocols and these classical BFT protocols: *metadata and data are written in separate quorum accesses*. Moreover, 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 2006, Attiya 2003, Chockler 2002, Gafni 2003, Jayanti 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. The 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 latter, 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 due to the fact that all known techniques require server-to-server communication [Cachin 2006], servers sending update notifications to clients [Martin 2002] or write-backs [Goodson 2004, Malkhi 1998b]. None of these mechanisms is implementable using general-purpose storage clouds.

To ensure confidentiality of stored data on the clouds without requiring a key distribution service, we employ a *secret sharing scheme* [Shamir 1979]. In this scheme, a special party called dealer distributes a secret to n players, but each player gets only a share of this secret. The main properties of the scheme is that at least $f+1 \leq n$ 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 receives just a share of the data being written, besides the metadata. This ensures that no individual cloud will have access to the data being stored, but that clients that have authorization to access the data will be granted access to the shares of (at least) $f+1$ different clouds and will be able to rebuild the original 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.

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 to not erase 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.

3.5 DEPSKY-A– Available DepSky

The first DEPSKY protocol is called DEPSKY-A, and improves the availability and integrity of cloud-stored data by replicating it on several providers using quorum techniques. Algorithm 1 presents this protocol. Due to space constraints we encapsulate some of the protocol steps in the functions of the first two rows of Table 1. 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.

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 (lines 12). 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 does its first writing in a data unit du (lines 3-5, max_ver_{du} is initialized as 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* just fetches the metadata files from a quorum of clouds (line 16), chooses the one with greatest version number (line 17) and reads the value corresponding to this version number and the cryptographic hash found in the chosen metadata (lines 19-22). After receiving the first reply that satisfies this condition the reader cancels the pending RPCs and returns the value (lines 22-24).

The rationale of why this protocol provides the desired properties is the following (proofs on the Appendix). Avail-

Function	Description
$queryMetadata(du)$	obtains the correctly signed file metadata stored in the container du of $n-f$ out-of the n 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. Blocks until it receives write confirmations from $n-f$ clouds.
$\mathsf{H}(value)$	returns the cryptographic hash of *value*.

Table 1. Functions used in the DEPSKY-A protocols.

Algorithm 1: DEPSKY-A

```
1  procedure DepSkyAWrite(du,value)
2  begin
3      if max_ver_du = 0 then
4          m ⟵ queryMetadata(du)
5          max_ver_du ⟵ max({m[i].ver : 0 ≤ i ≤ n−1})
6      new_ver ⟵ max_ver_du + 1
7      v[0 .. n−1] ⟵ value
8      writeQuorum(du,"value"+new_ver, v)
9      new_meta ⟵ ⟨new_ver, H(value)⟩
10     sign(new_meta, K_rw)
11     v[0 .. n−1] ⟵ new_meta
12     writeQuorum(du,"metadata", v)
13     max_ver_du ⟵ new_ver

14 function DepSkyARead(du)
15 begin
16     m ⟵ queryMetadata(du)
17     max_id ⟵ i : m[i].ver = max({m[i].ver : 0 ≤ i ≤ n−1})
18     v[0 .. n−1] ⟵ ⊥
19     parallel for 0 ≤ i < n−1 do
20         tmp_i ⟵ cloud_i.get(du, "value" +m[max_id].ver)
21         if H(tmp_i) = m[max_id].digest then v[i] ⟵ tmp_i
22     wait until ∃i : v[i] ≠ ⊥
23     for 0 ≤ i ≤ n−1 do cloud_i.cancel_pending()
24     return v[i]
```

ability 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.

3.6 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 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[1].

The DEPSKY-CA protocol integrates these techniques with the DEPSKY-A protocol (Algorithm 2). The additional cryptographic and coding functions needed are in Table 2.

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 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

Table 2. Functions used in the DEPSKY-CA protocols.

The DEPSKY-CA protocol is very similar to DEPSKY-A with the following differences: (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 30-31); (2.) the data stored in $cloud_i$ is composed by the share of the key $s[i]$ and the encoded block $e[i]$ (lines 12, 30-31); and (3.) $f+1$ replies are necessary to read the data unit's current value instead of one on DEPSKY-A (line 28). 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 the *digest* field of a metadata. If a key distribution infrastructure is available, or if readers and writer share a common key k, the secret sharing scheme can be removed (lines 7, 9 and 31 are not necessary).

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

Algorithm 2: DEPSKY-CA

```
1  procedure DepSkyCAWrite(du,value)
2  begin
3      if max_ver_du = 0 then
4          m ⟵ queryMetadata(du)
5          max_ver_du ⟵ max({m[i].version : 0 ≤ i ≤ n−1})
6      new_ver ⟵ max_ver_du + 1
7      k ⟵ generateSecretKey()
8      e ⟵ E(value, k)
9      s[0 .. n−1] ⟵ share(k, n, f+1)
10     v[0 .. n−1] ⟵ encode(e, n, f+1)
11     for 0 ≤ i < n−1 do
12         d[i] ⟵ ⟨s[i], e[i]⟩
13         h[i] ⟵ H(d[i])
14     writeQuorum(du,"value"+new_ver, d)
15     new_meta ⟵ ⟨new_ver, h⟩
16     sign(new_meta, K_rw)
17     v[0 .. n−1] ⟵ new_meta
18     writeQuorum(du,"metadata", v)
19     max_ver_du ⟵ new_ver

20 function DepSkyCARead(du)
21 begin
22     m ⟵ queryMetadata(du)
23     max_id ⟵ i : m[i].ver = max({m[i].ver : 0 ≤ i ≤ n−1})
24     d[0 .. n−1] ⟵ ⊥
25     parallel for 0 ≤ i ≤ n−1 do
26         tmp_i ⟵ cloud_i.get(du, "value" +m[max_id].ver)
27         if H(tmp_i) = m[max_id].digest[i] then d[i] ⟵ tmp_i
28     wait until |{i : d[i] ≠ ⊥}| > f
29     for 0 ≤ i ≤ n−1 do cloud_i.cancel_pending()
30     e ⟵ decode(d.e, n, f+1)
31     k ⟵ combine(d.s, n, f+1)
32     return D(e, k)
```

The rationale of the correctness of the protocol is similar to the one for DEPSKY-A (proofs also on the Appendix). 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.

3.7 Read Optimization

The DEPSKY-A algorithm described in Section 3.5 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: even using just a single value, the application will be paying 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 we obtain the desired value. 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., an 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.

3.8 Supporting Multiple Writers – Locks

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 backend 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, a coordination system like Zookeeper [Hunt 2010] or DepSpace [Bessani 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).

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. The protocol is the following:

1. A process p that wants to be a writer (and has permission to be), first lists files on the data unit container on all n clouds and tries to find a zero-byte file called *lock-ID-T*. If such file is found on a quorum of clouds, $ID \neq p$ and the local time t on the process is smaller than $T+\Delta$, being Δ a safety margin concerning the difference between the synchronized clocks of all writers, someone else is the writer and p will wait until $T+\Delta$.
2. If the test fails, p can write a lock file called *lock-p-T* on all clouds, being $T = t + writer\_lease\_time$.
3. In the last step, p lists again all files in the data unit container searching for other lock files with $t < T+\Delta$ besides the one it wrote. If such file is found, p 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. Otherwise, p becomes the single-writer for the data unit until T.

Several remarks can be made about this protocol. First, the last step 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. Moreover, unlocking can be easily done through the removal of the lock files. Third, the protocol requires synchronized clocks in order to employ leases and thus tolerate writer crashes. Finally, this lock protocol is only *obstruction-free* [Herlihy 2003]: if several process try to become writers at the same time, it is possible that none of them are successful. However, due to the backoff on step 3, this situation should be very rare on the envisioned deployments for the systems.

3.9 Additional Protocols

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. The creation of a data unit can be easily done through the invocation of 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 3.4, 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 too costly for the organization and thus some garbage collection is necessary. The protocol for doing that is very simple: a writer just lists all files *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. In this case we need a reconfiguration protocol to move the blocks from one cloud to another. The process is the following: (1.) the writer reads the data (probably from the other clouds and not from the one being removed); (2.) it 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 read this file and accept the reconfiguration if this file is read from at least $f+1$ clouds.

3.10 Dealing with Weakly Consistent Clouds

Both DEPSKY-A and DEPSKY-CA protocols implement *single-writer multi-reader regular registers* if the clouds being accessed provide *regular semantics*. However, several clouds do not ensure this guarantee, but instead provide *read-after-write* or *eventual consistency* [Vogels 2009] for the data stored (e.g., Amazon S3 [Ama 2010]).

With a slight modification, our protocols can work with these weakly consistent clouds. The modification is very

simple: repeat the data file reading from the clouds until the required condition is satisfied (receiving 1 or $f+1$ data units, respectively in lines 22 and 28 of Algorithms 1 and 2). This modification ensures the read of a value described on a read metadata will be repeated until it is available.

This modification makes the DEPSKY protocols be *consistency-proportional* in the following sense: if the underlaying clouds provide regular semantics, the protocols provide regular semantics; if the clouds provide read-after-write semantics, the protocol satisfies read-after-write semantics; and finally, if the clouds provide eventually consistency, the protocols are eventually consistent. Notice that if the underlying clouds are heterogeneous in terms of consistency guarantees, DEPSKY ensures the weakest consistency among those provided. This comes from the fact that reading of a recently write value depends on the reading of the new metadata file, which, after a write is complete, will only be available eventually on weakly consistent clouds.

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 steps 2 and 3 of the protocol. 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 is done in most Byzantine quorum protocols (e.g., [Malkhi 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.

4. 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 protocols; and (3) cloud providers drivers, which implement 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 is about 2910 lines of code, being 1122 lines for the drivers.

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 provider.

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 2008] and Jerasure [Plank 2007] were used for secret sharing and erasure code, respectively.

5. 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 is the advantage in terms of performance and availability of using replicated 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 are currently not many more than four cloud storage providers that are adequate for storing critical data. Our evaluation uses the following cloud storage providers with their default configurations: Amazon S3, Windows Azure, Nirvanix and Rackspace.

5.1 Economical

Storage cloud providers usually charge their users based on the amount of data uploaded, downloaded and stored on them. Table 3 presents the cost in US Dollars of executing 10,000 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 latter. All estimations presented on this section were calculated based on the values charged by the four clouds at September 25th, 2010.

In the table, the columns "DEPSKY-A", "DEPSKY-A opt", "DEPSKY-CA" e "DEPSKY-CA opt" present the costs of using the DEPSKY protocols with the read optimization respectively disabled and enabled. 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 $n = 4$ is roughly the sum of the costs of using the four clouds, as expected. However, if the read optimization is employed, the

Operation	DU Size	DEPSKY-A	DEPSKY-A opt	DEPSKY-CA	DEPSKY-CA opt	Amazon S3	Rackspace	Azure	Nirvanix
10kb 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
10kb Writes	100kb	0.60	0.60	0.30	0.30	0.14	0.08	0.09	0.29
	1Mb	6.16	6.16	3.08	3.08	1.46	0.78	0.98	2.93
	10Mb	61.5	61.5	30.8	30.8	14.6	7.81	9.77	29.3

Table 3. Estimated costs per 10000 operations (in US Dollars). DEPSKY-A and DEPSKY-CA costs are computed for the realistic case of 4 clouds ($f = 1$). The "DEPSKY-A opt" and "DEPSKY-CA opt" setups consider the cost-optimal version of the protocols with no failures.

less expensive cloud cost dominates the cost of executing reads (only one out-of four clouds is accessed in fault-free executions). For DEPSKY-CA, the cost of reading and writing 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 read also reduces this 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. Recall that the costs for the optimized versions of the protocol account only for the best case in terms of monetary costs: reads are executed on the required less expensive clouds. In the worst case, the more expensive clouds will be used instead.

The storage costs of a 1Mb data unit for different numbers of stored versions is presented in Figure 3. We present the curves only for one data unit size because other size costs are directly proportional.

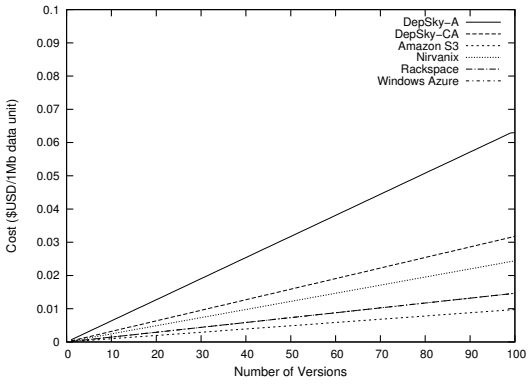


Figure 3. Storage costs of a 1Mb data unit for different numbers of stored versions.

The results depicted in the figure show that the cost of DEPSKY-CA storage 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. Notice that the metadata costs are almost irrelevant when compared with the data size since its size is less than 500 bytes.

5.2 PlanetLab deployment

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 then 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.

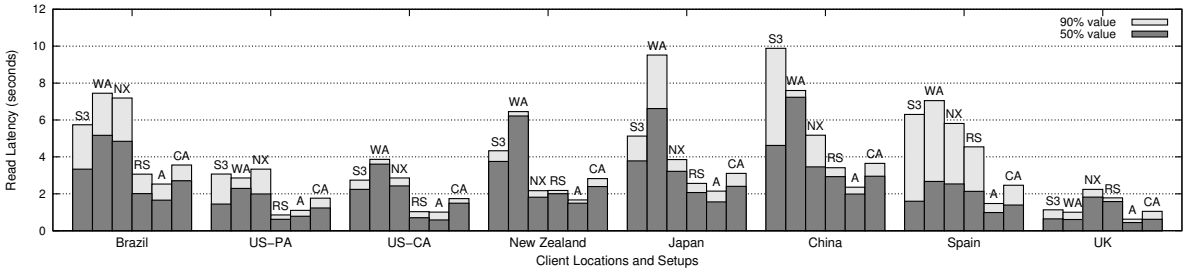
The logger application executes periodically a *measurement epoch*, which comprises: read the data unit (DU) from each of the clouds individually, one after another; read the DU using DEPSKY-A; read the DU using DEPSKY-CA; sleep until the next epoch. The goal is to read the data through different setups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

We deployed the logger on eight PlanetLab machines across the Internet, on four 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.

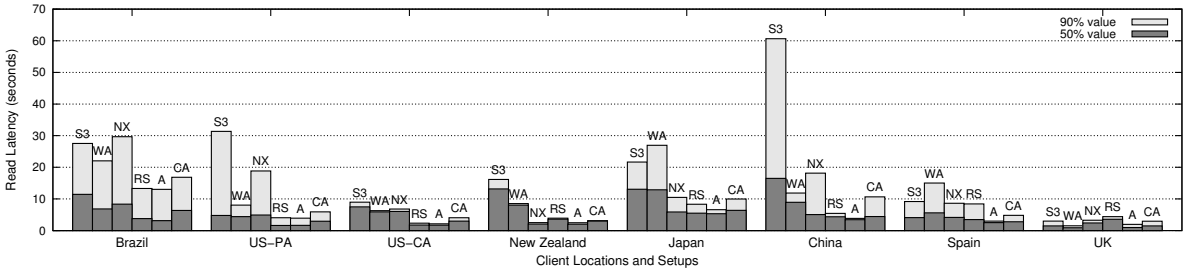
In the experiments, the local costs, in which the protocols incur due to the use of cryptography and erasure codes, are negligible for DEPSKY-A and account for at most 5% of the read and 10% of the write latencies on DEPSKY-CA.

Reads. Figure 4 presents the 50% and 90% percentile of all observed latencies of the reads executed (i.e., the values below which 50% and 90% of the observations fell). The number of reads executed on each site is presented on the second column of Table 5.

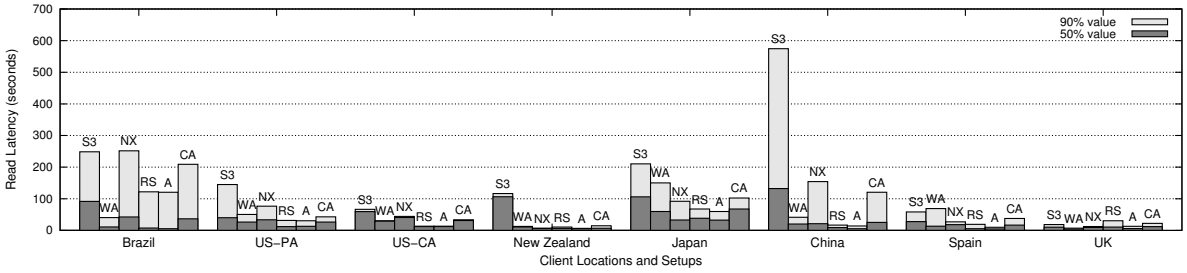
Based on the results presented in the figure, several points can be highlighted. First, DEPSKY-A presents the best latency of all but one setups. This is explained by the fact that it waits for 3 out-of 4 copies of the metadata but only one of the data, and it usually obtains it from the best cloud available during the execution. Second, DEPSKY-CA latency is closely related with the second best cloud storage provider,



(a) 100kb DU.



(b) 1Mb DU.



(c) 10Mb DU.

Figure 4. $50^{th}/90^{th}$-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 $f = 1$.

since it waits for at least 2 out-of 4 data blocks. Finally, notice that 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 highlight 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 that 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 3.7 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) 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, respectively.

Writes. We modified our logger application to execute writes instead of reads and deployed it on the same machines we executed the reads. We run it for two days in October and collected the logs, with at least 500 measurements for each location and data size. Due to space constraints, we do not present all these results, but illustrate the costs of write operations for different data sizes discussing only the observed results for an UK client. The 50% and 90% percentile of the latencies observed are presented in Figure 5.

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

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

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

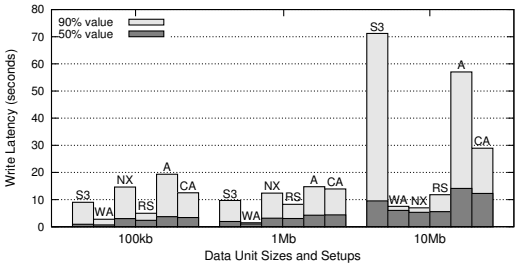


Figure 5. $50^{th}/90^{th}$-percentile latency (in seconds) for 100kb, 1Mb and 10Mb DU write operation for a PlanetLab client at the UK. The bar names are the same as in Figure 4. DEPSKY-A and DEPSKY-CA are configured with $n = 4$ and $f = 1$.

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 on all clouds, which consumes more bandwidth at the client side and requires replies from at least three clouds.

Secret sharing overhead. As discussed in Section 3.6, 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 31 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 protocols latency in the worst case[2].

Throughput. Table 4 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.

[2] For a more compreensive discussion about the overhead imposed by Java secret sharing see [Bessani 2008].

By the table it is possible to observe that the read throughput decreases from DEPSKY-A to DEPSKY-CA and then to S3, at the same time that write throughput increases for this same sequence. The higher read throughput of DEPSKY when compared with S3 is due to the fact that it fetches the data from all clouds on 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 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 5 presents the *perceived availability* of all setups calculated as $\frac{reads\_completed}{reads\_tried}$ from different locations.

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 5. First, DEPSKY-A and DEPSKY-CA are the two single setups that presented an availability of 1.0000 in almost all

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

Table 5. The perceived availability of all setups evaluated from different points of the Internet.

locations[3]. 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. This is a fact that companies moving critical applications to the cloud have to be fully aware.

6. Related Work

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 (which is substantially different than providing write-once, read-maybe archival storages such as the one described in [Storer 2007]).

There are many protocols for Byzantine quorums systems for register implementation (e.g., [Goodson 2004, Hendricks 2007, Malkhi 1998a, Martin 2002]), however, few of them address the model in which servers are passive entities that do not run protocol code [Abraham 2006, Attiya 2003, Jayanti 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 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 2006] presents a single-writer multi-reader regular register construction that requires two communication round-trips both for reading and writing in absence of contention. 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 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 2006, Goodson 2004, Hendricks 2007], but none of them mention the use of secret sharing or the provision of confidentiality. 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.

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 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 backup [Vrable 2009], implement a database [Brantner 2008] or add provenance to the stored data [Muniswamy-Reddy 2010]. However none of these works provide guarantees like confidentiality and availability and do not consider 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 with DEPSKY. The HAIL (High-Availability Integrity Layer) protocol set [Bowers 2009] aggregates 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 (opposite to DEPSKY, that uses the storage clouds as they are), and does not provide guarantee of confidentiality of the stored data. The RACS (Redundant Array of Cloud Storage) system [Abu-Libdeh 2010] employs RAID5-like techniques (mainly erasure codes) to implement high-available and storage-efficient data replication on diverse clouds. Differently from DEPSKY, the RACS system does not try to solve security problems of cloud storage, but instead deals with "economic failures" and vendor lock-in.

[3] 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.

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 cloud replication works present an experimental evaluation with diverse clouds as it is presented in this paper.

There are several works about obtaining trustworthiness from untrusted clouds. Depot improves the resilience of cloud storage making similar assumptions to DEPSKY, that storage clouds are fault-prone black boxes [Mahajan 2010]. 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 2010] and Venus [Shraer 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.

7. Conclusion

This paper presents the design and evaluation of DEPSKY, a storage service that improves the availability and confidentiality provided by commercial storage cloud services. 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 cloud providers. 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 clouds.

The paper also presents an extensive evaluation of the system. The key conclusion is that it provides confidentiality and improved availability at a cost roughly double of using a single cloud for a practical scenario, which seems to be a good compromise for critical applications.

Acknowledgments

We warmly thank our shepherd Scott Brandt and the EuroSys'11 reviewers for their comments on earlier versions of this paper. This work was partially supported by the EC through project TCLOUDS (FP7/2007-2013, ICT-257243), and by the FCT through the ReD (PTDC/EIA-EIA/109044/2008), RC-Clouds (PCT/EIA-EIA/115211/2009) and the Multiannual and CMU-Portugal Programmes.

References

[Ama 2010] Amazon S3 FAQ: What data consistency model does amazon S3 employ? `http://aws.amazon.com/s3/faqs/`, 2010.

[tcl 2010] Project TCLOUDS – trustworthy clouds - privacy and resilience for Internet-scale critical infrastructure. `http://www.tclouds-project.eu/`, 2010.

[Ehs 2010] UK NHS Systems and Services. `http://www.connectingforhealth.nhs.uk/`, 2010.

[Abraham 2006] Ittai Abraham, Gregory Chockler, Idit Keidar, and Dahlia Malkhi. Byzantine disk Paxos: optimal resilience with Byzantine shared memory. *Distributed Computing*, 18(5):387–408, April 2006.

[Abu-Libdeh 2010] Hussam Abu-Libdeh, Lonnie Princehouse, and Hakim Weatherspoon. RACS: A case for cloud storage diversity. *Proc. of the 1st ACM Symposium on Cloud Computing*, pages 229–240, June 2010.

[Attiya 2003] Hagit Attiya and Amir Bar-Or. Sharing memory with semi-Byzantine clients and faulty storage servers. In *Proc. of the 22rd IEEE Symposium on Reliable Distributed Systems - SRDS 2003*, pages 174–183, October 2003.

[Bessani 2008] Alysson N. Bessani, Eduardo P. Alchieri, Miguel Correia, and Joni S. Fraga. DepSpace: a Byzantine fault-tolerant coordination service. In *Proc. of the 3rd ACM European Systems Conference – EuroSys'08*, pages 163–176, April 2008.

[Bowers 2009] Kevin D. Bowers, Ari Juels, and Alina Oprea. HAIL: a high-availability and integrity layer for cloud storage. In *Proc. of the 16th ACM Conference on Computer and Communications Security - CCS'09*, pages 187–198, 2009.

[Brantner 2008] Matthias Brantner, Daniela Florescu, David Graf, Donald Kossmann, and Tim Kraska. Building a database on S3. In *Proc. of the 2008 ACM SIGMOD International Conference on Management of Data*, pages 251–264, 2008.

[Cachin 2006] Christian Cachin and Stefano Tessaro. Optimal resilience for erasure-coded Byzantine distributed storage. In *Proc. of the Int. Conference on Dependable Systems and Networks - DSN 2006*, pages 115–124, June 2006.

[Chockler 2009] Gregory Chockler, Rachid Guerraoui, Idit Keidar, and Marko Vukolić. Reliable distributed storage. *IEEE Computer*, 42(4):60–67, 2009.

[Chockler 2002] Gregory Chockler and Dahlia Malkhi. Active disk Paxos with infinitely many processes. In *Proc. of the 21st Symposium on Principles of Distributed Computing – PODC'02*, pages 78–87, 2002.

[Feldman 2010] Ariel J. Feldman, William P. Zeller, Michael J. Freedman, and Edward W. Felten. SPORC: Group collaboration using untrusted cloud resources. In *Proc. of the 9th USENIX Symposium on Operating Systems Design and Implementation – OSDI'10*, pages 337–350, October 2010.

[Gafni 2003] Eli Gafni and Leslie Lamport. Disk Paxos. *Distributed Computing*, 16(1):1–20, 2003.

[Ghemawat 2003] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The Google file system. In *Proc. of the 19th ACM Symposium on Operating Systems Principles – SOSP'03*, pages 29–43, 2003.

[Gibson 1998] Garth Gibson, David Nagle, Khalil Amiri, Jeff Butler, Fay Chang, Howard Gobioff, Charles Hardin, Erik Riedel, David Rochberg, and Jim Zelenka. A cost-effective, high-bandwidth storage architecture. In *Proc. of the 8th Int. Con-*

ference on Architectural Support for Programming Languages and Operating Systems - ASPLOS'98, pages 92–103, 1998.

[Goodson 2004] Garth Goodson, Jay Wylie, Gregory Ganger, and Micheal Reiter. Efficient Byzantine-tolerant erasure-coded storage. In *Proc. of the Int. Conference on Dependable Systems and Networks - DSN'04*, pages 135–144, June 2004.

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

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

[Hendricks 2007] James Hendricks, Gregory Ganger, and Michael Reiter. Low-overhead byzantine fault-tolerant storage. In *Proc. of the 21st ACM Symposium on Operating Systems Principles – SOSP'07*, pages 73–86, 2007.

[Henry 2009] Alyssa Henry. Cloud storage FUD (failure, uncertainty, and durability). Keynote Address at the 7th USENIX Conference on File and Storage Technologies, February 2009.

[Herlihy 2003] Maurice Herlihy, Victor Lucangco, and Mark Moir. Obstruction-free syncronization: double-ended queues as an example. In *Proc. of the 23th IEEE Int. Conference on Distributed Computing Systems - ICDCS 2003*, pages 522–529, July 2003.

[Hunt 2010] Patrick Hunt, Mahadev Konar, Flavio Junqueira, and Benjamin Reed. Zookeeper: Wait-free coordination for Internet-scale services. In *Proc. of the USENIX Annual Technical Conference – ATC 2010*, pages 145–158, June 2010.

[Jayanti 1998] Prasad Jayanti, Tushar Deepak Chandra, and Sam Toueg. Fault-tolerant wait-free shared objects. *Journal of the ACM*, 45(3):451–500, May 1998.

[Krawczyk 1993] Hugo Krawczyk. Secret sharing made short. In *Proc. of the 13th Int. Cryptology Conference – CRYPTO'93*, pages 136–146, August 1993.

[Lamport 1986] Leslie Lamport. On interprocess communication (part II). *Distributed Computing*, 1(1):203–213, January 1986.

[Lamport 1982] Leslie Lamport, Robert Shostak, and Marshall Pease. The Byzantine generals problem. *ACM Transactions on Programing Languages and Systems*, 4(3):382–401, July 1982.

[Liskov 2006] Barbara Liskov and Rodrigo Rodrigues. Tolerating Byzantine faulty clients in a quorum system. In *Proc. of the 26th IEEE Int. Conference on Distributed Computing Systems - ICDCS'06*, July 2006.

[Mahajan 2010] Prince Mahajan, Srinath Setty, Sangmin Lee, Allen Clement, Lorenzo Alvisi, Mike Dahlin, and Michael Walfish. Depot: Cloud storage with minimal trust. In *Proc. of the 9th USENIX Symposium on Operating Systems Design and Implementation – OSDI 2010*, pages 307–322, October 2010.

[Malkhi 1998a] Dahlia Malkhi and Michael Reiter. Byzantine quorum systems. *Distributed Computing*, 11(4):203–213, 1998.

[Malkhi 1998b] Dahlia Malkhi and Michael Reiter. Secure and scalable replication in Phalanx. In *Proc. of the 17th IEEE Symposium on Reliable Distributed Systems - SRDS'98*, pages 51–60, October 1998.

[Martin 2002] Jean-Philippe Martin, Lorenzo Alvisi, and Mike Dahlin. Minimal Byzantine storage. In *Proc. of the 16th Int. Symposium on Distributed Computing – DISC 2002*, pages 311–325, 2002.

[McCullough 2010] John C. McCullough, JohnDunagan, Alec Wolman, and Alex C. Snoeren. Stout: An adaptive interface to scalable cloud storage. In *Proc. of the USENIX Annual Technical Conference – ATC 2010*, pages 47–60, June 2010.

[Metz 2009] Cade Metz. DDoS attack rains down on Amazon cloud. *The Register*, October 2009. `http://www.theregister.co.uk/2009/10/05/amazon_bitbucket_outage/`.

[Muniswamy-Reddy 2010] Kiran-Kumar Muniswamy-Reddy, Peter Macko, and Margo Seltzer. Provenance for the cloud. In *Proc. of the 8th USENIX Conference on File and Storage Technologies – FAST'10*, pages 197–210, 2010.

[Naone 2009] Erica Naone. Are we safeguarding social data? Technology Review published by MIT Review, `http://www.technologyreview.com/blog/editors/22924/`, February 2009.

[Plank 2007] James S. Plank. Jerasure: A library in C/C++ facilitating erasure coding for storage applications. Technical Report CS-07-603, University of Tennessee, September 2007.

[Rabin 1989] Michael Rabin. Efficient dispersal of information for security, load balancing, and fault tolerance. *Journal of the ACM*, 36(2):335–348, February 1989.

[Sarno 2009] David Sarno. Microsoft says lost sidekick data will be restored to users. *Los Angeles Times*, Oct. 15th 2009.

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

[Shamir 1979] Adi Shamir. How to share a secret. *Communications of ACM*, 22(11):612–613, November 1979.

[Shraer 2010] Alexander Shraer, Christian Cachin, Asaf Cidon, Idit Keidar, Yan Michalevsky, and Dani Shaket. Venus: Verification for untrusted cloud storage. In *Proc. of the ACM Cloud Computing Security Workshop – CCSW'10*, 2010.

[Storer 2007] Mark W. Storer, Kevin M. Greenan, Ethan L. Miller, and Kaladhar Voruganti. Potshards: Secure long-term storage without encryption. In *Proc. of the USENIX Annual Technical Conference – ATC 2007*, pages 143–156, June 2007.

[Vogels 2009] Werner Vogels. Eventually consistent. *Communications of the ACM*, 52(1):40–44, 2009.

[Vrable 2009] Michael Vrable, Stefan Savage, and Geoffrey M. Voelker. Cumulus: Filesystem backup to the cloud. *ACM Transactions on Storage*, 5(4):1–28, 2009.

[Vukolic 2010] Marko Vukolic. The Byzantine empire in the intercloud. *ACM SIGACT News*, 41(3):105–111, 2010.

[Weil 2006] Sage A. Weil, Scott A. Brandt, Ethan L. Miller, Darrell D. E. Long, and Carlos Maltzahn. Ceph: A scalable, high-performance distributed file system. In *Proc. of the 7th USENIX Symposium on Operating Systems Design and Implementation – OSDI 2006*, pages 307–320, 2006.

Protocols Correctness Proofs

This section presents correctness proofs of the DEPSKY-A and DEPSKY-CA protocols. The first lemma states that the auxiliary functions presented in Table 1 are wait-free.

Lemma 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 are 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).

Lemma 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. DepSkyCAWrite) 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 $n-f$ servers, from which at least $n-f-f \geq f+1$ are correct. ■

Lemma 3 *The outstanding metadata obtained on an 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 before 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 1 *All* DEPSKY *read and write operations are wait-free operations.*

Proof: Both Algorithms 1 and 2 use functions *queryMetadata* and *writeQuorum*. As show in Lemma 1, these operations can not block. Besides that, read operations make processes wait for the value associated with the outstanding metadata. Lemma 2 states that there are at least $f+1$ correct servers with this data, and thus at least one of them will answer the RPC of lines 20 and 27 of Algorithms 1 and 2, respectively, with values that matches 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 last two theorems show that DEPSKY-A and DEPSKY-CA implement single-writer multi-reader regular registers.

Theorem 2 *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 3 states that the outstanding metadata obtained on line lines 16-17 of Algorithm 1 corresponds to the last write executed or one of the writes being executed. Lemma 2 states that the value associated with this metadata is available from at least $f+1$ correct servers, and thus it can be obtained by the client on lines 19-22: just a single valid reply will suffice for releasing the barrier of line 22 and return the value. ■

Theorem 3 *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 with the one for DEPSKY-A. Lemma 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 2 states that the values associated with this metadata are stored on at least $f+1$ non-faulty clouds, and thus a reader can obtain them through the execution of lines 25-28: 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 30-32), inverting the processing done by the writer on DepSkyCAWrite (lines 7-10). ■