EURECOM
Department of Networking and Security
Campus SophiaTech
CS 50193
06904 Sophia Antipolis cedex
FRANCE

Research Report RR-13-291

Hybris: Consistency Hardening in Robust Hybrid Cloud Storage

October 12th, 2013
Last update January 30th, 2014

Dan Dobre†, Paolo Viotti⋆ and Marko Vukolić⋆

†NEC Labs Europe & ⋆EURECOM

Tel : (+33) 4 93 00 81 00 — Fax : (+33) 4 93 00 82 00
Email : dan.dobre@neclab.eu
{paolo.viotti, marko.vukolic}@eurecom.fr

[1]EURECOM's research is partially supported by its industrial members: BMW Group Research & Technology, IABG, Monaco Telecom, Orange, SAP, SFR, ST Microelectronics, Swisscom, Symantec.

Hybris: Consistency Hardening in Robust Hybrid Cloud Storage

Dan Dobre[†], Paolo Viotti[⋆] and Marko Vukolić[⋆]

[†]NEC Labs Europe & [⋆]EURECOM

Abstract

We present Hybris key-value store, the first robust hybrid cloud storage system. Hybris robustly replicates metadata on trusted private premises (private cloud), separately from data which is replicated across multiple untrusted public clouds. Hybris introduces a technique we call *consistency hardening* which consists in leveraging strong metadata consistency to guarantee to Hybris applications strong data consistency (linearizability) without entailing any modifications to weakly (e.g., eventually) consistent public clouds, which actually store data. Moreover, Hybris efficiently and robustly tolerates up to f potentially malicious clouds. Namely, in the common case, Hybris writes replicate data across $f + 1$ clouds, whereas reads involve a single cloud. In the worst case, f additional clouds are used.

We evaluate Hybris using a series of micro and macrobenchmarks and show that Hybris significantly outperforms comparable multi-cloud storage systems and approaches the performance of bare-bone commodity public cloud storage.

Index Terms

consistency hardening, efficiency, hybrid cloud storage, multi cloud storage, strong consistency.

Contents

List of Figures

1 Introduction

Hybrid cloud storage entails storing data on private premises as well as on one (or more) remote, public cloud storage providers. To enterprises, such hybrid design brings the best of both worlds: the benefits of public cloud storage (e.g., elasticity, flexible payment schemes and disaster-safe durability) as well as the control over enterprise data. In a sense, hybrid cloud eliminates to a large extent the concerns that companies have with entrusting their data to commercial clouds [1] — as a result, enterprise-class hybrid cloud storage solutions are booming with all leading storage providers, such as EMC[2], IBM[3], Microsoft[4] and others, offering their proprietary solutions.

As an alternative approach to addressing trust and reliability concerns associated with public cloud storage providers, several research works (e.g., [5, 4, 25]) considered storing data *robustly* into public clouds, by leveraging *multiple* commodity cloud providers. In short, the idea behind these public *multi-cloud* storage systems such as DepSky [5], ICStore [4] and SPANStore [25] is to leverage multiple cloud providers with the goals of distributing the trust across clouds, increasing reliability, availability and performance, and/or addressing vendor lock-in concerns (e.g., cost).

However, the existing robust multi-cloud storage systems suffer from serious limitations. In particular, the robustness of these systems does not concern consistency: namely, these systems provide consistency that is at best proportional [5] to that of the underlying clouds which very often provides only eventual consistency [23]. Moreover, these storage systems scatter storage metadata across public clouds increasing the difficulty of storage management and impacting performance. Finally, despite the benefits of the hybrid cloud approach, none of the existing robust storage systems considered leveraging resources on private premises (e.g., in companies and even in many households).

In this paper, we unify the hybrid cloud approach with that of robust multi-cloud storage and present Hybris, the first robust hybrid cloud storage system. The key idea behind Hybris is that it keeps all storage *metadata* on private premises, even when those metadata pertain to data outsourced to public clouds. This separation of metadata from data allows Hybris to significantly outperform existing robust public multi-cloud storage systems, both in terms of system performance (e.g., latency) and storage cost, while providing strong consistency guarantees. The salient features of Hybris are as follows:

- *Consistency Hardening:* Hybris is a multi-writer multi-reader key-value storage system that guarantees linearizability (atomicity) [16] of reads and writes even in presence of eventually consistent public clouds [23]. To this end, Hybris employs a novel scheme we call *consistency hardening*: Hybris leverages the atomicity of metadata stored locally on premises to mask the possible inconsistencies of data stored at public clouds.

- *Robustness to malicious clouds:* Hybris puts no trust in any given public cloud provider; namely, Hybris can mask arbitrary (including malicious) faults of up to f public clouds. Interestingly, Hybris relies on as few as $f+1$ clouds in the common case (when the system is synchronous and without faults), using up to f additional clouds in the worst case (e.g., network partitions, cloud inconsistencies and faults). This is in sharp contrast to existing multi-cloud storage systems that involve up to $3f+1$ clouds to mask f malicious ones (e.g., [5]).

[1] See e.g., `http://blogs.vmware.com/consulting/2013/09/the-snowden-leak-a-windfall-for-hybrid-cloud.html`.

[2] `http://www.emc.com/campaign/global/hybridcloud/`.

[3] `http://www.ibm.com/software/tivoli/products/hybrid-cloud/`.

[4] `http://www.storsimple.com/`.

- *Efficiency:* Hybris is efficient and incurs low cost. In common case, a Hybris write involves as few as $f + 1$ public clouds, whereas reads involve only a single cloud, despite the fact that clouds are untrusted. Hybris achieves this without relying on expensive cryptographic primitives; indeed, in masking malicious faults, Hybris relies solely on cryptographic hashes.

Clearly, for Hybris to be truly robust, it has also to replicate metadata reliably. Given inherent trust in private premises, we assume faults within private premises that can affect Hybris metadata to be crash-only. To maintain the Hybris footprint small and to facilitate its adoption, we chose to replicate Hybris metadata layering Hybris on top of Apache ZooKeeper coordination service [17]. Hybris clients act simply as ZooKeeper clients — our system does not entail any modifications to ZooKeeper, hence facilitating Hybris deployment. In addition, we designed Hybris metadata service to be easily portable to SQL-based replicated RDBMS as well as NoSQL data stores that export conditional update operation (e.g., HBase or MongoDB), which can then serve as alternatives to ZooKeeper.

Finally, Hybris optionally supports caching of data stored at public clouds, as well as symmetric-key encryption for data confidentiality leveraging trusted Hybris metadata to store and share cryptographic keys.

We implemented Hybris in Java and evaluated it using both microbenchmarks and the YCSB [9] macrobenchmark. Our evaluation shows that Hybris significantly outperforms state-of-the-art robust multi-cloud storage systems, with a fraction of the cost and stronger consistency.

The rest of the paper is organized as follows. In § 2, we present the Hybris architecture and system model. Then, in § 3, we give the algorithmic aspects of the Hybris protocol. In § 4 we discuss Hybris implementation and optimizations. In § 5 we present Hybris performance evaluation. We overview related work in § 6, and conclude in § 7.

2 Hybris overview

Hybris architecture. High-level design of Hybris is given in Figure 1. Hybris mixes two types of resources: 1) private, trusted resources that consist of computation and (limited) storage resources and 2) public (and virtually unlimited) untrusted storage resources in the clouds. Hybris is designed to leverage commodity public cloud storage repositories whose API does not offer computation, i.e., key-value stores (e.g., Amazon S3).

Hybris stores metadata separately from public cloud data. Metadata is stored within the key component of Hybris called Reliable MetaData Service (RMDS). RMDS has no single point of failure and, in our implementation, resides on private premises.

On the other hand, Hybris stores data (mainly) in untrusted public clouds. Data is replicated across multiple cloud storage providers for robustness, i.e., to mask cloud outages and even malicious faults. In addition to storing data in public clouds, Hybris architecture supports data caching on private premises. While different caching solutions exist, our Hybris implementation reuses Memcached[5], an open source distributed caching system.

Finally, at the heart of the system is the Hybris client, whose library is responsible for interactions with public clouds, RMDS and the caching service. Hybris clients are also responsible for encrypting and decrypting data in case data confidentiality is enabled — in this case, clients leverage RMDS for sharing encryption keys (see Sec. 3.7).

In the following, we first specify our system model and assumptions. Then we define Hybris data model and specify its consistency and liveness semantics.

[5] `http://memcached.org/`.

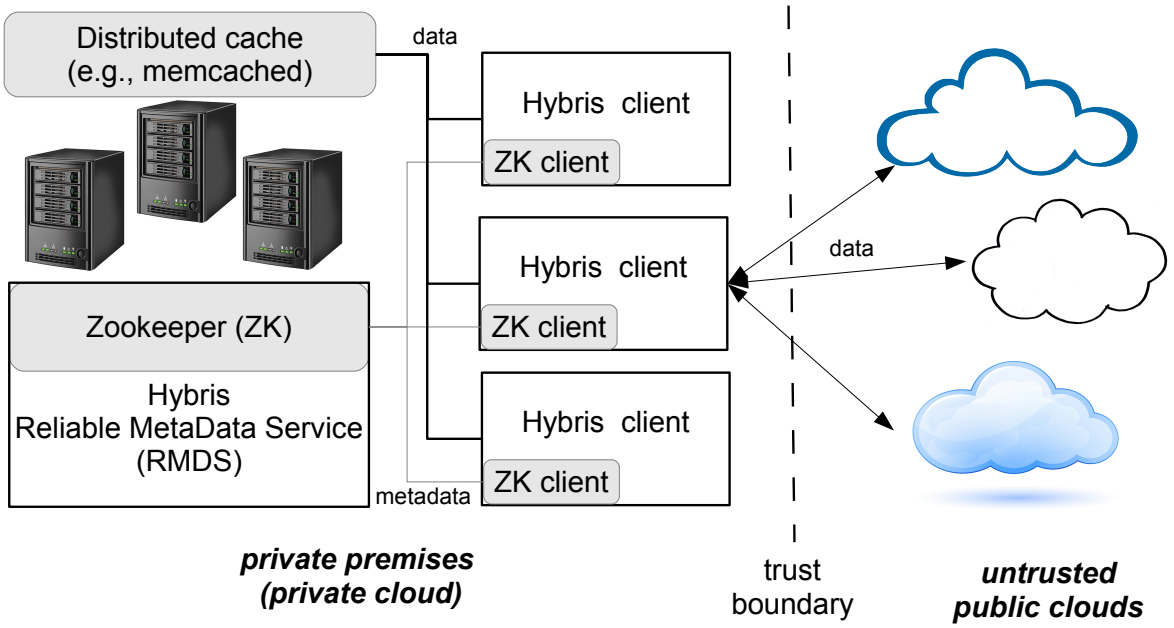


Figure 1: Hybris architecture. Reused (open-source) components are depicted in grey.

System model. We assume an unreliable distributed system where any of the components might fail. In particular, we consider dual fault model, where: (i) the processes on private premises (i.e., in the private cloud) can fail by crashing, and (ii) we model public clouds as potentially malicious (i.e., arbitrary-fault prone [20]) processes. Processes that do not fail are called *correct*.

Processes on private premises are clients and metadata servers. We assume that *any* number of clients and any minority of metadata servers can be (crash) faulty. Moreover, we allow up to f public clouds to be (arbitrary) faulty; to guarantee Hybris availability, we require at least $2f+1$ public clouds in total. However, Hybris consistency is maintained regardless of the number of public clouds.

Similarly to our fault model, our communication model is dual, with the model boundary coinciding with our trust boundary (see Fig. 1).[6] Namely, we assume that the communication among processes located in the private portion of the cloud is partially synchronous [12] (i.e., with arbitrary but finite periods of asynchrony), whereas the communication among clients and public clouds is entirely asynchronous (i.e., does not rely on any timing assumption) yet reliable, with messages between correct clients and clouds being eventually delivered.

Our consistency model is likewise dual. We model processes on private premises as classical state machines, with their computation proceeding in indivisible, atomic steps. On the other hand, we model clouds as eventually consistent [23]; roughly speaking, eventual consistency guarantees that, if no new updates are made to a given data item, eventually all accesses to that item will return the last updated value.

Finally, for simplicity, we assume an adversary that can coordinate malicious processes as well as process crashes. However, we assume that the adversary cannot subvert cryptographic hash functions we use (SHA-1), and that it cannot spoof the communication among non-malicious processes.

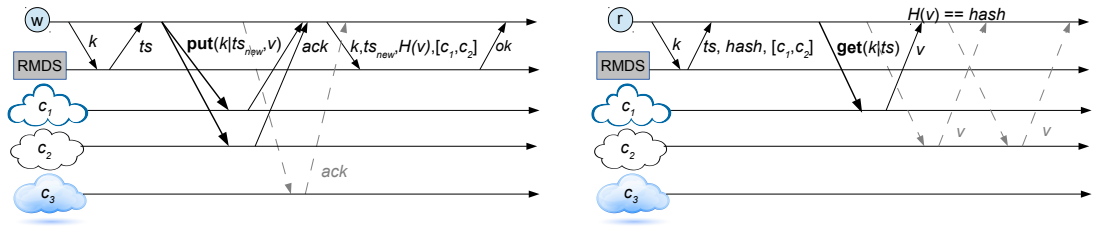
[6] We believe that our dual fault and communication models reasonably model the typical hybrid cloud deployment scenarios.

Hybris data model and semantics. Similarly to commodity public cloud storage services, Hybris exports a key-value store (KVS) API; in particular, Hybris address space consists of flat containers, each holding multiple keys. The KVS API features four main operations: *(i)* PUT$(cont, key, value)$, to put *value* under *key* in container *cont*; *(ii)* GET$(cont, key, value)$, to retrieve the value; DELETE$(cont, key)$ to remove the respective entry and *(iv)* LIST$(cont)$ to list the keys present in container *cont*. We collectively refer to Hybris operations that modify storage state (e.g., PUT and DELETE) as *write* operations, whereas the other operations (e.g., GET and LIST) are called *read* operations.

Hybris implements a multi-writer multi-reader key-value storage. Hybris is strongly consistent, i.e., it implements atomic (or *linearizable* [16]) semantics. In distributed storage context, atomicity provides an illusion that a complete operation *op* is executed instantly at some point in time between its invocation and response, whereas the operations invoked by faulty clients appear either as complete or not invoked at all.

Despite providing strong consistency, Hybris is highly available. Hybris writes by a correct client are guaranteed to eventually complete [15]. On the other hand, Hybris guarantees a read operation by a correct client to complete always, except in an obscure corner case where there is an infinite number of writes to the same key concurrent with the read operation (this is called finite-write termination [1]).

3 Hybris Protocol



(a) PUT (k, v) (b) GET (k)

Figure 2: Hybris PUT and GET protocol illustration ($f = 1$). Common-case communication is depicted in solid lines.

3.1 Overview

The key component of Hybris is its RMDS component which maintains metadata associated with each key-value pair. In the vein of Farsite [3], Hybris RMDS maintains pointers to data locations and cryptographic hashes of the data. However, unlike Farsite, RMDS additionally includes a client-managed logical timestamp for concurrency control, as well as data size.

Such Hybris metadata, despite being lightweight, is powerful enough to enable tolerating arbitrary cloud failures. Intuitively, the cryptographic hash within a trusted and consistent RMDS enables end-to-end integrity protection: it ensures that neither corrupted values produced by malicious clouds, nor stale values retrieved from inconsistent clouds, are ever returned to the application. Complementarily, data size helps prevent certain denial-of-service attack vectors by a malicious cloud (see Sec. 4.2).

Furthermore, Hybris metadata acts as a directory pointing to $f + 1$ clouds that have been previously updated, enabling a client to retrieve the correct value despite f of them being arbitrary faulty. In fact, with Hybris, as few as $f + 1$ clouds are sufficient to ensure both consistency

and availability of read operations (namely GET) — indeed, Hybris GET *never involves more than* $f+1$ *clouds* (see Sec. 3.3). Additional f clouds (totaling $2f+1$ clouds) are only needed to guarantee that write operations (namely PUT) are available as well (see Sec. 3.2). Note that since f clouds can be faulty, and a value needs to be stored in $f+1$ clouds for durability, overall $2f+1$ clouds are required for PUT operations to be available in the presence of f cloud outages.

Finally, besides cryptographic hash and pointers to clouds, metadata includes a timestamp that, roughly speaking, induces a partial order of operations which captures the real-time precedence ordering among operations (atomic consistency). The subtlety of Hybris (see Sec. 3.5 for details) is in the way it combines timestamp-based lock-free multi-writer concurrency control within RMDS with garbage collection (Sec. 3.4) of stale values from public clouds to save on storage costs.

In the following we detail each Hybris operation individually.

3.2 put Protocol

Hybris PUT protocol entails a sequence of consecutive steps illustrated in Figure 2(a). To write a value v under key k, a client first fetches from RMDS the latest authoritative timestamp ts by requesting the metadata associated with key k. Timestamp ts is a tuple consisting of a sequence number sn and a client id cid. Based on timestamp ts, the client computes a new timestamp ts_{new}, whose value is $(sn+1, cid)$. Next, the client combines the key k and timestamp ts_{new} to a new key $k_{new} = k|ts_{new}$ and invokes **put** (k_{new}, v) on $f+1$ clouds in parallel. Concurrently, the clients starts a timer whose expiration is set to typically observed upload latencies (for a given value size). In the common case, the $f+1$ clouds reply to the the client in a timely fashion, before the timer expires. Otherwise, the client invokes **put** (k_{new}, v) on up to f secondary clouds (see dashed arrows in Fig. 2(a)). Once the client has received acks from $f+1$ different clouds, it is assured that the PUT is durable and proceeds to the final stage of the operation.

In the final step, the client attempts to store in RMDS the metadata associated with key k, consisting of the timestamp ts_{new}, the cryptographic hash $H(v)$, size of value v $size(v)$, and the list ($cloudList$) of pointers to those $f+1$ clouds that have acknowledged storage of value v. Notice, that since this final step is the linearization point of PUT it has to be performed in a specific way as discussed below.

Namely, if the client performs a straightforward update of metadata in RMDS, then it may occur that stored metadata is overwritten by metadata with a *lower* timestamp (old-new inversion), breaking the timestamp ordering of operations and Hybris consistency. To solve the old-new inversion problem, we require RMDS to export an atomic conditional update operation. Then, in the final step of Hybris PUT, the client issues conditional update to RMDS which updates the metadata for key k *only if* the written timestamp ts_{new} is *greater* than the timestamp for key k that RMDS already stores. In Section 4 we describe how we implement this functionality over Apache ZooKeeper API; alternatively other NoSQL and SQL DBMSs that support conditional updates can be used.

3.3 get in the common case

Hybris GET protocol is illustrated in Figure 2(b). To read a value stored under key k, the client first obtains from RMDS the latest metadata, comprised of timestamp ts, cryptographic hash h, value size s, as well a list $cloudList$ of pointers to $f+1$ clouds that store the corresponding value. Next, the client selects the first cloud c_1 from $cloudList$ and invokes **get** $(k|ts)$ on c_1, where $k|ts$ denotes the key under which the value is stored. Besides requesting the value, the client starts a timer set to the typically observed download latency from c_1 (given the value size s) (for that particular cloud). In the common case, the client is able to download the correct

value from the first cloud c_1 in a timely manner, before expiration of its timer. Once it receives value v, the client checks that v hashes to hash h comprised in metadata (i.e., if $H(v) = h$). If the value passes the check, then the client returns the value to the application and the GET completes.

In case the timer expires, or if the value downloaded from the first cloud does not pass the hash check, the client sequentially proceeds to downloading the data from the second cloud from *cloudList* (see dashed arrows in Fig. 2(b)) and so on, until the client exhausts all $f + 1$ clouds from *cloudList*.[7]

In specific corner cases, caused by concurrent garbage collection (described in Sec. 3.4), failures, repeated timeouts (asynchrony), or clouds' inconsistency, the client has to take additional actions in GET (described in Sec. 3.5).

3.4 Garbage Collection

The purpose of garbage collection is to reclaim storage space by deleting obsolete versions of keys from clouds while allowing read and write operations to execute concurrently. Garbage collection in Hybris is performed by the writing client asynchronously in the background. As such, the PUT operation can give back control to the application without waiting for completion of garbage collection.

To perform garbage collection for key k, the client retrieves the list of keys prefixed by k from each cloud as well as the latest authoritative timestamp ts. This involves invoking $list(k|*)$ on every cloud and fetching metadata associated with key k from RMDS. Then for each key k_{old}, where $k_{old} < k|ts$, the client invokes DELETE (k_{old}) on every cloud.

3.5 get in the worst-case: Consistency Hardening

In the context of cloud storage, there are known issues with weak, e.g., eventual [23] consistency. With eventual consistency, even a correct, non-malicious cloud might deviate from atomic semantics (strong consistency) and return an unexpected value, typically a stale one. In this case, sequential common-case reading from $f + 1$ clouds as described in Section 3.3 might not return a value since a hash verification might fail at all $f + 1$ clouds. In addition to the case of inconsistent clouds, this anomaly may also occur if: (i) timers set by the client for a otherwise non-faulty cloud expire prematurely (i.e., in case of asynchrony or network outages), and/or (ii) values read by the client were concurrently garbage collected (Sec. 3.4).

To cope with these issues and eventual consistency in particular, Hybris introduces *consistency hardening*: namely, we leverage metadata service consistency to mask data inconsistencies in the clouds. Roughly speaking, with consistency hardening Hybris client indulgently reiterates the GET by reissuing a **get** to all clouds in parallel, and waiting to receive at least one value matching the desired hash. However, due to possible concurrent garbage collection (Sec. 3.4), a client needs to make sure it always compares the values received from clouds to the most recent key metadata. This can be achieved in two ways: (i) by simply looping the entire GET including metadata retrieval from RMDS, or (ii) by looping only **get** operations at $f + 1$ clouds while fetching metadata from RMDS only when metadata actually changes.

In Hybris, we use the second approach. Notice that this suggests that RMDS must be able to inform the client proactively about metadata changes. This can be achieved by having a RMDS that supports subscriptions to metadata updates, which is possible to achieve in, e.g..,

[7] As we discuss in details in Section 4, in our implementation, clouds in *cloudList* are ranked by the client by their typical latency in the ascending order, i.e., when reading the client will first read from the "fastest" cloud from *cloudList* and then proceed to slower clouds.

Apache ZooKeeper (using the concepts of *watches*, see Sec. 4 for details). The entire protocol executed only is common-case GET fails (Sec. 3.3) proceeds as follows:

1. A client first reads key k metadata from RMDS (i.e., timestamp ts, hash h, size s and cloud list *cloudList*) and subscribes for updates for key k metadata with RMDS.
2. Then, a client issues a parallel **get** $(k|ts)$ at all $f+1$ clouds from *cloudList*.
3. When a cloud $c \in$*cloudList* responds with value v_c, the client verifies $H(v_c)$ against h[8].
   (a) If the hash verification succeeds, the GET returns v_c.
   (b) Otherwise, the client discards v_c and reissues **get** $(k|ts)$ at cloud c.
4. At any point in time, if the client receives a metadata update notification for key k from RMDS, the client cancels all pending downloads, and repeats the procedure by going to step 1.

The complete Hybris GET, as described above, ensures finite-write termination [1] in presence of eventually consistent clouds. Namely, a GET may fail to return a value only theoretically, in case of infinite number of concurrent writes to the same key, in which case the garbage collection at clouds (Sec. 3.4) might systematically and indefinitely often remove the written values before the client manages to retrieve them.[9]

3.6 delete and list

Besides PUT and GET, Hybris exports the additional functions: DELETE and LIST— here, we only briefly sketch how these functions are implemented.

Both DELETE and LIST are local to RMDS and do not access public clouds. To delete a value, the client performs the PUT protocol with a special *cloudList* value $\perp$ denoting the lack of a value. Deleting a value creates metadata tombstones in RMDS, i.e. metadata that lacks a corresponding value in cloud storage. On the other hand, Hybris LIST simply retrieves from RMDS all keys associated with a given container *cont* and filters out deleted (tombstone) keys.

3.7 Confidentiality

Adding confidentiality to Hybris is straightforward.To this end, during a PUT, just before uploading data to $f+1$ public clouds, the client encrypts the data with a symmetric cryptographic key k_{enc}. Then, in the final step of the PUT protocol (see Sec. 3.2), when the client writes metadata to RMDS using conditional update, the client simply adds k_{enc} to metadata and computes the hash on ciphertext (rather than on cleartext). The rest of the PUT protocol remains unchanged. The client may generate a new key with each new encryption, or fetch the last used key from the metadata service, at the same time it fetches the last used timestamp.

To decrypt data, a client first obtains the most recently used encryption key k_{enc} from metadata retrieved from RMDS during a GET. Then, upon the retrieved ciphertext from some cloud successfully passes the hash test, the client decrypts data using k_{enc}.

[8] For simplicity, we model the absence of a value as a special NULL value that can be hashed.

[9] Notice that it is straightforward to modify Hybris to guarantee read availability even in case of an infinite number of concurrent writes, by switching off the garbage collection.

4 Implementation

We implemented Hybris in Java. The implementation pertains solely to the Hybris client side since the entire functionality of the metadata service (RMDS) is layered on top of Apache ZooKeeper client. Namely, Hybris does not entail any modification to the ZooKeeper server side. Our Hybris client is lightweight and consists of 2030 lines of Java code. Hybris client interactions with public clouds are implemented by wrapping individual native Java SDK clients (drivers) for each particular cloud storage provider[10] into a common lightweight interface that masks the small differences across native client libraries.

In the following, we first discuss in details our RMDS implementation with Zookeper API. Then, we describe several Hybris optimizations that we implemented.

4.1 ZooKeeper-based RMDS

We layered Hybris implementation over Apache ZooKeeper [17]. In particular, we durably store Hybris metadata as ZooKeeper *znodes*; in ZooKeeper znodes are data objects addressed by *paths* in a hierarchical namespace. In particular, for each instance of Hybris, we generate a root znode. Then, the metadata pertaining to Hybris container *cont* is stored under ZooKeeper path $\langle root \rangle / cont$. In principle, for each Hybris key k in container *cont*, we store a znode with path $path_k = \langle root \rangle / cont / k$.

ZooKeeper exports a fairly modest API to its applications. The ZooKeeper API calls relevant to us here are: (i) **create/setData**$(p, data)$, which creates/updates znode with path p containing *data*, (ii) **getData**(p) to retrieve data stores under znode with p, and (iii) **sync**(), which synchronizes a ZooKeeper replica that maintains the client's session with ZooKeeper leader. Only reads that follow after **sync**() will be atomic.[11]

Besides data, znodes have some specific Zookepeer metadata (not be confused with Hybris metadata which we store in znodes). In particular, our implementation uses znode version number vn, that can be supplied as an additional parameter to **setData** operation which then becomes a *conditional update* operation which updates znode only if its version number exactly matches vn.

Hybris put. At the beginning of PUT (k, v), when client fetches the latest timestamp ts for k, the Hybris client issues a **sync**() followed by **getData**$(path_k)$ to ensure an atomic read of ts. This **getData** call returns, besides Hybris timestamp ts, the internal version number vn of the znode $path_k$ which the client uses when writing metadata md to RMDS in the final step of PUT.

In the final step of PUT, the client issues **setData**$(path_k, md, vn)$ which succeeds only if the znode $path_k$ version is still vn. If the ZooKeeper version of $path_k$ changed, the client retrieves the new authoritative Hybris timestamp ts_{last} and compares it to ts. If $ts_{last} > ts$, the client simply completes a PUT (which appears as immediately overwritten by a later PUT with ts_{last}). In case, $ts_{last} < ts$, the client retries the last step of PUT with ZooKeeper version number vn_{last} that corresponds to ts_{last}. This scheme (inspired by [7]) is guaranteed to terminate since only a finite number of concurrent PUT operations use a timestamp smaller than ts.

Hybris get. In interacting with RMDS during GET, Hybris client simply needs to make sure its metadata is read atomically. To this end, a client always issues a **sync**() followed by **get-**

[10] Currently, Hybris supports Amazon S3, Google Cloud Storage, Rackspace Cloud Files and Windows Azure.

[11] Without **sync**, ZooKeeper may return stale data to client, since reads are served locally by ZooKeeper replicas which might have not yet received the latest update.

Data($path_k$), just like in our PUT protocol. In addition, for subscriptions for metadata updates in GET (Sec. 3.5) we use the concept of ZooKeeper *watches* (set by e.g., **getData**) which are subscriptions on znode update notifications. We use these notifications in Step 4 of the algorithm described in Section 3.5.

4.2 Optimizations

Cloud latency ranks. In our Hybris implementation, clients rank clouds by latency and prioritize clouds with lower latency. Hybris client then uses these cloud latency ranks in common case to: (i) write to $f+1$ clouds with the lowest latency in PUT, and (ii) to select from *cloudList* the cloud with the lowest latency as *preferred* cloud in GET. Initially, we implemented the cloud latency ranks by reading once (i.e., upon initialization of the Hybris client) a default, fixed-size (100kB) object from each of the public clouds. Interestingly, during our experiments, we observed that the cloud latency rank significantly varies with object size as well as the type of the operation (PUT vs. GET). Hence, our implementation establishes several cloud latency ranks depending on the file size and the type of operation. In addition, Hybris client can be instructed to refresh these latency ranks when necessary.

Preventing "Big File" DoS attacks. A malicious preferred cloud may mount a DoS attack against Hybris client during a read by sending, instead of the correct file, a file of arbitrary length. In this way, a client would not detect a malicious fault until computing a hash of the received file. To cope with this attack, Hybris client uses value size s that Hybris stores and simply cancels the downloads whose payload size extends over s.

Caching. Our Hybris implementation enables data caching on private portion of the system. We implemented simple write-through cache and caching-on-read policies. With write-through caching enabled, Hybris client simply writes to cache in parallel to writing to clouds. On the other hand, with caching-on-read enabled, Hybris client upon returning a GET value to the application, writes lazily the GET value to the cache. In our implementation, we use Memcached distributed cache that exports a key-value interface just like public clouds. Hence, all Hybris writes to the cache use exactly the same addressing as writes to public clouds (i.e., using **put**($k|ts, v$)). To leverage cache within a GET, Hybris client upon fetching metadata always tries first to read data from the cache (i.e., by issuing **get** ($k|ts$) to Memcached), before proceeding normally with a GET.

5 Evaluation

For evaluation purposes, we deployed Hybris "private" components (namely, Hybris client, metadata service (RMDS) and cache) as virtual machines (VMs) within an OpenStack[12] cluster that acts as our private cloud located in Sophia Antipolis, France. Our OpenStack cluster consists of: two master nodes running on a dual quad-core Xeon L5320 server clocked at 1.86GHz, with 16GB of RAM, two 1TB hardware RAID5 volumes, and two 1Gb/s network interfaces; and worker nodes that execute on six dual exa-core Xeon E5-2650L servers clocked at 1.8GHz, with 128GB of RAM, ten 1TB disks and four 1Gb/s network cards.[13] We use the KVM hypervisor, and each machine in the physical cluster runs the Grizzly release of OpenStack on top of a Ubuntu 12.04.2 Linux distribution.

[12] `http://www.openstack.org/`.

[13] Our hardware and network configuration closely resembles the one suggested by commercial private cloud providers, such as Rackspace.

We collocate ZooKeeper and Memcached (in their default configurations) using three VMs of the aforementioned private cloud. Each VM has one quad-core virtual processor clocked at 2.40GHz, 4GB of RAM, one PATA virtual hard drive and it is connected to the others through a gigabit Ethernet network. All VMs run the Ubuntu Linux 13.10 distribution images, updated with the most recent patches.

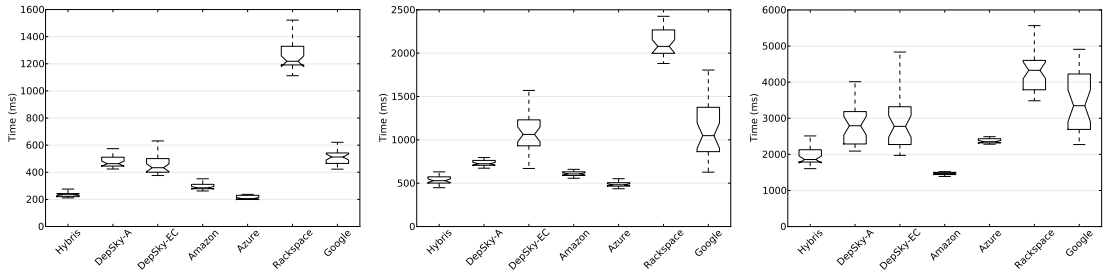
In addition, several OpenStack VMs with same characteristics are used for running the client instances. Each VM has 100Mb/s internet connectivity for both upload and download bandwidths. Clients are configured to interact with four cloud providers: Amazon S3, Rackspace CloudFiles, Microsoft Azure (all located in Europe) and Google Cloud Storage (in US).

We evaluated Hybris performance in several experiments that focus on the arguably most interesting case where $f = 1$ [10], i.e., where at most one public cloud can exhibit arbitrary faults.

Experiment 1: Common-case latency. In this experiment, we benchmark the common-case latency of Hybris with respect to those of DepSky-A [5],[14] DepSky-EC (i.e. a version of DepSky featuring erasure codes support), and the four individual clouds underlying Hybris and DepSky, namely Amazon, Azure, Rackspace and Google. For this microbenchmark we perform a set of independent PUT and GET operations for sizes ranging from 100kB to 50MB and output the median latencies together with 95% confidence intervals on boxplot graphs.

We repeated each experiment 30 times, and each set of GET and PUT operations has been performed one after the other in order to avoid side effects due to internet routing and traffic fluctuations.

In Figures 3 and 4 we show latency boxplots of the clients as we vary the size of the object to be written or read.[15] We observe that Hybris GET latency (Fig. 3) closely follows those of the fastest cloud provider, as in fact it downloads the object from that specific cloud, according to Hybris cloud latency ranks (see Sec. 4). The difference between the fastest clouds and Hybris GET is slightly more pronounced for larger files (e.g., 10MB) due to time Hybris needs to perform hash integrity check which linearly increases with object size. We further observe (Fig. 4) that Hybris roughly performs as fast as the second fastest cloud storage provider. This is expected since Hybris uploads to clouds are carried out in parallel to the first two cloud providers previously ranked by their latency.



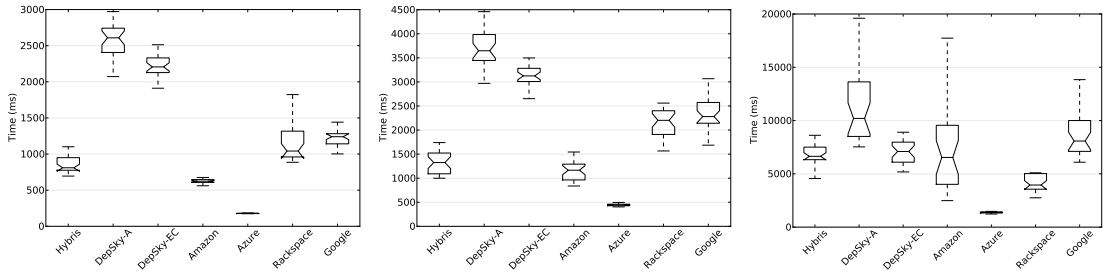
(a) 100kB GET (b) 1MB GET (c) 10MB GET

Figure 3: Latencies of GET operations.

It is worth noting that Hybris outperforms DepSky-A and DepSky-EC in both PUT and GET operations. The difference is significant in particularly for smaller to medium object sizes (100kB

[14]We used open-source DepSky implementation available at `https://code.google.com/p/depsky/`.

[15]In the boxplots the central line is showing the median, the box corresponds to 1^{st} and 3^{rd} quartiles while whiskers are drawn at the most extreme data points within 1.5 times the interquartile range from 1^{st} and 3^{rd} quartiles.



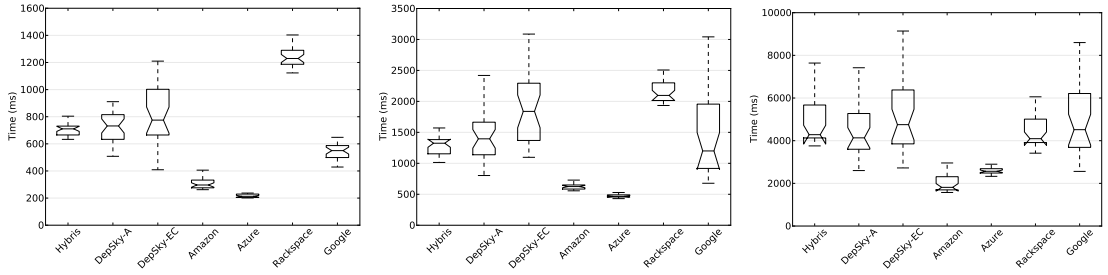
(a) 100kB PUT (b) 1MB PUT (c) 10MB PUT

Figure 4: Latencies of PUT operations.

and 1MB). This is explained by the fact that Hybris stores metadata locally, whereas DepSky needs to fetch metadata across clouds. With increasing file sizes (10MB) network latency takes over and the difference is less pronounced in particular since DepSky-EC erasure codes data and uploads some 25% less data than Hybris.

Throughout the tests, we observed a significant variance in cloud performance and in particular for uploading large objects at Amazon. This variance was confirmed in several repetitions of the experiment.

Experiment 2: Latency under faults. In order to assess the impact of faulty clouds on Hybris GET performance, we repeat Experiment 1 with one cloud serving tampered objects. This experiment aims at stress testing the common case optimization of Hybris to download objects from a single cloud. In particular, we focused on the worst case for Hybris, that is, we injected the fault on the closest cloud, i.e. the one likely to be chosen for the download because of its low latency. The failure injection has been carried out by manually tampering the data to be subsequently downloaded by the clients.



(a) 100kB GET (b) 1MB GET (c) 10MB GET

Figure 5: Latencies of GET operations with one faulty cloud.

Figure 5 shows the download times of Hybris, DepSky-A and DepSky-EC for objects of different sizes, as well as those of individual clouds, for reference. Hybris performance is nearly the sum of the download times by the two fastest clouds, as the GET downloads in this case sequentially. However, despite its single cloud read optimization, Hybris performance under faults remains comparable to that of DepSky variants that download objects in parallel.

Experiment 3: Throughput scalability. The aim of this experiment is to test the scalability limits of Hybris. To assess the usage of the available bandwidth by Hybris we run several Hybris

clients simultaneously, each uploading a different object, thus not generating any collisions on RMDS. In the experiment, clients are collocated and share the network bandwidth.

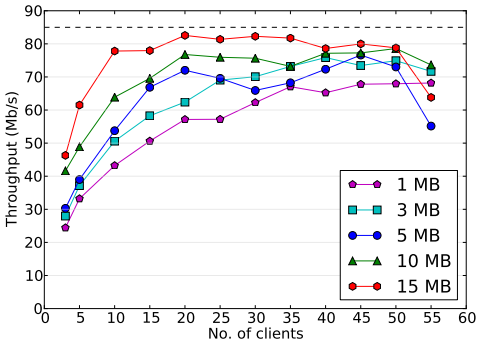


Figure 6: Aggregated throughput of Hybris clients performing PUT operations.

As Figure 5 shows, a relatively small number of Hybris clients can fully saturate network upload bandwidth, which in this case was of about 85Mb/s. Since the number of clients needed to saturate the internet throughput is small when clients share upload bandwidth, in the next experiment we modify the scalability experiment to isolate Hybris RMDS and test its scalability limits.

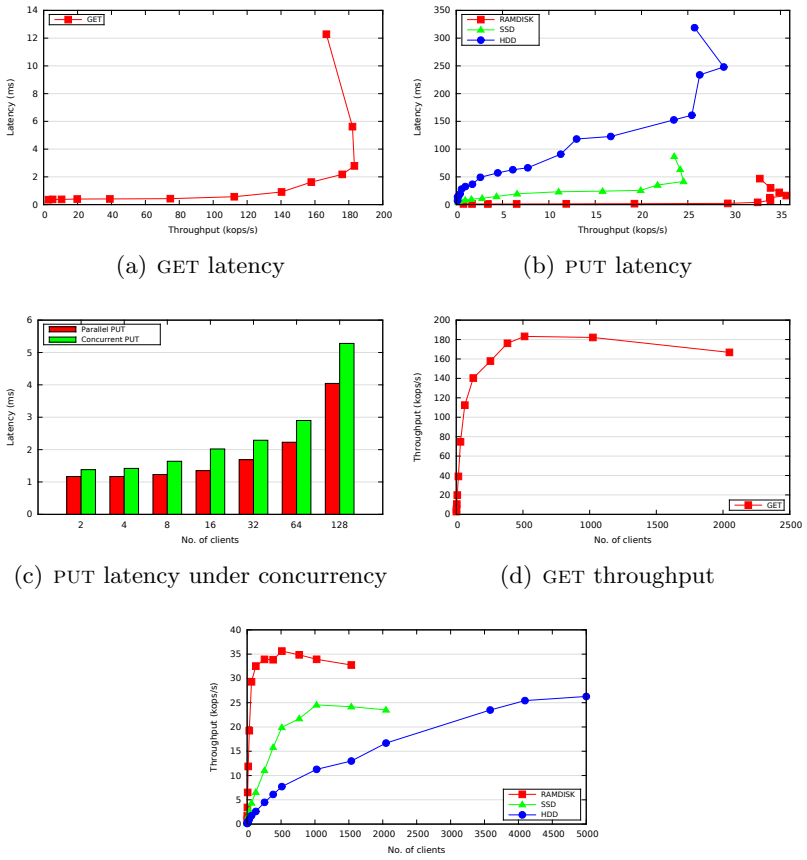
Experiment 4: RMDS performance. In this experiment we stress our ZooKeeper-based RMDS implementation in order to assess its performance when the links to clouds are not the bottleneck. For this purpose, we short-circuit public clouds and simulate upload by writing a 100 byte payload to an in-memory hash map. To mitigate possible performance impact of the shared OpenStack cloud we perform (only) this experiment deploying RMDS on a dedicated cluster of three 8-core Xeon E3-1230 V2 machines (3.30GHz, 20 GB ECC RAM, 1GB Ethernet, 128GB SATA SSD, 250 GB SATA HDD 10000rpm). The obtained results concerning metadata reads and writes performance are shown in Figure 7.

Figure 7(a) shows GET latency as we increase throughput. The observed peak throughput of roughly 180 kops/s achieved with latencies below 4 ms is due to the fact that syncing reads in ZooKeeper comes with a modest overhead and we take advantage of read locality in ZooKeeper to balance requests across ZooKeeper nodes. Furthermore, since RMDS has a small footprint, all read requests are serviced directly from memory without incurring the cost of stable storage access.

On the other hand, PUT operations incur the expense of atomic broadcast within ZooKeeper and stable storage accesses in the critical path. Figure 7(b) shows the latency-throughput curve for three different classes of stable storage backing ZooKeeper, namely conventional HDD, SSD and RAMDISK, which would be replaced by non-volatile RAM in a production-ready system. The observed differences suggest that the choice of stable storage for RMDS is crucial for overall system performance, with HDD-based RMDS incurring latencies nearly one order of magnitude higher than RAMDISK-based at peak throughput of 28 kops/s (resp. 35 kops/s). As expected, SSD-based RMDS is in the middle of the latency spectrum spanned by the other two storage types.

To understand the impact of concurrency on RMDS performance, we evaluated the latency of PUT under heavy contention to a single key. Figure 7(c) shows that despite 128 clients writing concurrently to the same key, which in our view represents a conservative concurrency upper bound, the latency overhead incurred is only 30% over clients writing to separate keys.

Finally, Figures 7(d) and 7(e) depict throughput growth curves as more clients performing operations in closed-loop are added to the system. Specifically 7(d) suggests that ZooKeeper-based RMDS is able to service read requests coming from $2K$ clients near peak throughput. On the other hand, Figure 7(e) shows again the performance discrepancy when using different stable storage types, with RAMDISK and HDD at opposite ends of the spectrum. Observe that HDD peak throughput, despite being below that of RAMDISK, slightly overtakes SSD throughput with $5K$ clients.



(a) GET latency

(b) PUT latency

(c) PUT latency under concurrency

(d) GET throughput

(e) PUT throughput

Figure 7: Performance of metadata read and write operations.

Experiment 5: Caching. In this experiment we test caching in Hybris which is configured to implement both write-through and caching-on-read policies. We configured Memcached with 128 MB cache limit and with 10MB single object limit. In our experiment we varied blob sizes from 1kB to 10 MB and measure average latency. The Experiment workload is YCSB workload B (95% reads, 5% writes). The results for GET with and without caching are depicted in Figure 8.

We can observe that caching decreases Hybris latency by an order of magnitude when cache is large enough compared to object size. As expected, the benefits of cache diminish with increase in cache misses. This experiment shows that Hybris can very simply benefit from caching, unlike other multi-cloud storage protocols (see also Table 2).

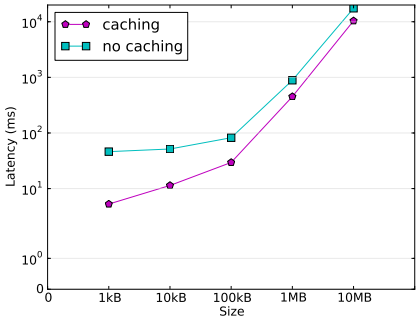


Figure 8: Hybris GET latency with YCSB workload B.

System	PUT	GET	Storage Cost / Month	Total
ICStore [4]	60	376	570	1006
DepSky-A [5]	30	376	285	691
DepSky-EC [5]	30	196	142	368
Hybris	10	120	190	320
Amazon S3	5	120	95	220

Table 1: Cost of cloud storage systems in USD for 10^6 transactions and 1MB files, totaling to 1TB of storage.

Cost comparison. Table 1 summarizes the monetary costs incurred by several cloud storage systems in the common case (i.e. synchrony, no failures, no concurrency), including Amazon S3 as the baseline. For the purpose of calculating costs, given in USD, we set $f = 1$ and assume a symmetric workload that involves 10^6 PUT (i.e. modify) and 10^6 GET operations accessing $1MB$ files totaling to $1TB$ of storage over the period of 1 month. This corresponds to a modest workload of roughly 40 hourly operations. We abstract away the cost of private cloud infrastructure in Hybris, and assume that such infrastructure is available on premises ahead of time. Further the cost per transaction, storage, and outbound traffic are taken from Amazon S3 as of 10/12/2013. The basis for costs calculation is Table 2.

We observe that Hybris overhead is twice the baseline both for PUT and storage because Hybris stores data in 2 clouds in the common case. Since Hybris touches a single cloud once for each GET operation, the cost of GET equals that of the baseline, and hence is optimal.

6 Related Work

Multi-cloud storage systems. Several storage systems (e.g., [2, 6, 25, 5, 4]) have used multiple clouds in boosting data robustness, notably reliability and availability. Early multi-cloud systems such as RACS [2] and HAIL [6] assumed immutable data, hence not addressing any concurrency aspects.

Multi-cloud storage systems closest to Hybris are DepSky [5] and ICStore [4]. For clarity, we overview main aspects of these three systems in Table 2. ICStore is a cloud storage system that models cloud faults as outages and implements robust access to shared data. Hybris advantages over ICStore include tolerating malicious clouds and smaller storage blowup[16]. On the other hand, DepSky considers malicious clouds, yet requires $3f+1$ clouds, unlike Hybris. Furthermore, DepSky consistency guarantees are weaker than those of Hybris, even when clouds behave as

[16]Blowup of a given redundancy scheme is defined as the ratio between the total storage size needed to store redundant copies of a file, over the original unreplicated file size.

Protocol	Semantics		Common case performance	
	Cloud faults	**Consistency**	**No. of Cloud operations**	**Blow-up**
ICStore [4]	crash-only	atomic[1]	$(4f+2)(D+m)$ (writes) $(2f+1)(D+m)$ (reads)	$4f+2$
DepSky [5]	**arbitrary**	regular[1]	$(2f+1)(D+m)$ (writes) $(2f+1)(D+m)$ (reads)	$2f+1$[2]
Hybris	**arbitrary**	**atomic**	$(\mathbf{f+1})\mathbf{D}$ (writes) $\mathbf{1D}$ (reads)	$\mathbf{f+1}$

[1]Unlike Hybris, to achieve atomic (resp., regular) semantics, ICStore (resp., DepSky) require public clouds to be atomic (resp., regular).

[2]DepSky also implements an erasure coding variant which features $\frac{2f+1}{f+1}$ storage blowup.

Table 2: Comparison of existing robust multi-writer cloud storage protocols. We distinguish cloud data operations (D) from cloud metadata operations (m).

strongly consistent. Finally, the distinctive feature of Hybris is consistency hardening which guarantees Hybris atomicity even in presence of eventually consistent clouds, which may harm the consistency guarantees of both ICStore and DepSky.

Finally, SPANStore [25] is a recent multi-cloud storage system that seeks to minimize the cost of use of multi-cloud storage. However, the reliability of SPANStore is considerably below that of Hybris. Namely, SPANStore is not robust, as it features a centralized cloud placement manager which is a single point of failure. Furthermore, SPANStore considers crash-only clouds and uses leased lock-based writes.

Separating data from metadata. Separating metadata from data is not a novel idea in distributed systems. For example, Farsite [3] is an early protocol that tolerates malicious faults by replicating metadata (e.g., cryptographic hashes and directory) separately from data. Hybris builds upon these techniques yet, unlike Farsite, Hybris implements multi-writer/multi-reader semantics and is robust against timing failures as it relies on lock-free concurrency control rather than locks (or leases). Furthermore, unlike Farsite, Hybris supports ephemeral clients and has no server code, targeting commodity cloud APIs.

Separation of data from metadata is intensively used in crash-tolerant protocols. For example in the Hadoop Distributed File System (HDFS), modeled after the Google File System [14], HDFS NameNode is responsible for maintaining metadata, while data is stored on HDFS DataNodes.

Other notable crash-tolerant storage systems that separate metadata from data include LDR [13] and BookKeeper [18]. LDR [13] implements asynchronous multi-writer multi-reader read/write storage and, like Hybris, uses pointers to data storage nodes within its metadata and requires $2f+1$ data storage nodes. However, unlike Hybris, LDR considers full-fledged servers as data storage nodes and tolerates only their crash faults. BookKeeper [18] implements reliable single-writer multi-reader shared storage for logs. BookKeeper stores metadata on servers (bookies) and data (i.e., log entries) in log files (ledgers). Like in Hybris RMDS, bookies point to ledgers, facilitating writes to $f+1$ ledgers and reads from a single ledger in common-case. However, Hybris differs significantly from BookKeeper: namely, Hybris supports multiple writers, tolerates malicious faults of data repositories and is designed with different deployment environment and applications in mind.

Interestingly, all crash-tolerant protocols related to Hybris, that separate metadata from data (e.g., [13, 18], but also Gnothi [24]), need $2f+1$ data repositories in the worst case, just like our Hybris which tolerates arbitrary faults.

Finally, the idea of separating control and data planes in systems tolerating arbitrary faults was used also in [26] in the context of replicated state machines (RSM). While the RSM approach of [26] could obviously be used for implementing storage as well, Hybris proposes a far more scalable and practical solution, while also tolerating pure asynchrony across data communication links, unlike [26].

Systems based on trusted components. Several systems have used trusted hardware components to reduce the overhead of replication despite malicious faults from $3f + 1$ to $2f + 1$ replicas, typically in the context of RSM (e.g., [11, 8, 19, 22]). Some of these systems, like CheapBFT [19], employ only $f + 1$ replicas in the common case.

Conceptually, Hybris is similar to these systems in that Hybris relies on trusted hardware and uses $2f + 1$ trusted metadata replicas (needed for ZooKeeper) and $2f + 1$ (untrusted) clouds. However, compared to these systems, Hybris is novel in several ways. Most importantly, existing systems typically entail placing a trusted hardware component within an untrusted process, which raises concerns over practicality of such an approach. In contrast, Hybris trusted hardware (private cloud) exists separately from untrusted processes (public clouds), with this model (of a hybrid cloud) being in fact inspired by actual practical system deployments. Moreover, Hybris focuses on storage rather than on generic RSM and offers a practical, deployment-ready solution.

7 Conclusion and Future Work

In this paper we presented Hybris, the first robust hybrid storage system. Hybris replicates data across multiple untrusted and possibly inconsistent public clouds, while it replicates metadata within trusted premises of a private cloud. This design allows *consistency hardening* — Hybris leverages strong consistency of metadata stored off-clouds to mask the weak consistency of data stored in clouds and turn it into strong consistency. Hybris tolerates up to f arbitrary public cloud faults and is very efficient: Hybris write accesses only $f + 1$ clouds in the synchronous, failure-free common-case, while a Hybris read accesses a single, "closest" cloud. In the worst case, f additional clouds are used. Hybris is modular and reuses open-source software. Our evaluation confirms the practicality of our system.

In future work, we plan to extend Hybris to support erasure coding schemes to further reduce storage blowup. Our system architecture does not prevent such erasure coding variants of Hybris that, however induce numerous tradeoffs [21] and require deeper insight.

References

[1] Ittai Abraham, Gregory Chockler, Idit Keidar, and Dahlia Malkhi. Byzantine Disk Paxos: Optimal Resilience with Byzantine Shared Memory. *Distributed Computing*, 18(5):387–408, 2006.

[2] Hussam Abu-Libdeh, Lonnie Princehouse, and Hakim Weatherspoon. RACS: a case for cloud storage diversity. In *SoCC*, pages 229–240, 2010.

[3] Atul Adya, William J. Bolosky, Miguel Castro, Gerald Cermak, Ronnie Chaiken, John R. Douceur, Jon Howell, Jacob R. Lorch, Marvin Theimer, and Roger P. Wattenhofer. Farsite: Federated, available, and reliable storage for an incompletely trusted environment. *SIGOPS Oper. Syst. Rev.*, 36(SI):1–14, December 2002.

[4] Cristina Basescu, Christian Cachin, Ittay Eyal, Robert Haas, Alessandro Sorniotti, Marko Vukolić, and Ido Zachevsky. Robust data sharing with key-value stores. In *Proceedings of DSN*, pages 1–12, 2012.

[5] Alysson Neves Bessani, Miguel P. Correia, Bruno Quaresma, Fernando André, and Paulo Sousa. Depsky: dependable and secure storage in a cloud-of-clouds. In *EuroSys*, pages 31–46, 2011.

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

[7] Gregory Chockler, Dan Dobre, and Alexander Shraer. Brief announcement: Consistency and complexity tradeoffs for highly-available multi-cloud store. In *The International Symposium on Distributed Computing (DISC)*, 2013.

[8] Byung-Gon Chun, Petros Maniatis, Scott Shenker, and John Kubiatowicz. Attested append-only memory: making adversaries stick to their word. In *SOSP*, pages 189–204, 2007.

[9] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking cloud serving systems with ycsb. In *SoCC*, pages 143–154, 2010.

[10] James C. Corbett, Jeffrey Dean, Michael Epstein, Andrew Fikes, Christopher Frost, J. J. Furman, Sanjay Ghemawat, Andrey Gubarev, Christopher Heiser, Peter Hochschild, Wilson C. Hsieh, Sebastian Kanthak, Eugene Kogan, Hongyi Li, Alexander Lloyd, Sergey Melnik, David Mwaura, David Nagle, Sean Quinlan, Rajesh Rao, Lindsay Rolig, Yasushi Saito, Michal Szymaniak, Christopher Taylor, Ruth Wang, and Dale Woodford. Spanner: Google's globally distributed database. *ACM Trans. Comput. Syst.*, 31(3):8, 2013.

[11] Miguel Correia, Nuno Ferreira Neves, and Paulo Veríssimo. How to tolerate half less one Byzantine nodes in practical distributed systems. In *SRDS*, pages 174–183, 2004.

[12] Cynthia Dwork, Nancy Lynch, and Larry Stockmeyer. Consensus in the presence of partial synchrony. *J. ACM*, 35(2):288–323, April 1988.

[13] Rui Fan and Nancy Lynch. Efficient Replication of Large Data Objects. In *Proceedings of DISC*, pages 75–91, 2003.

[14] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The google file system. In *SOSP*, pages 29–43, 2003.

[15] Maurice Herlihy. Wait-Free Synchronization. *ACM Trans. Program. Lang. Syst.*, 13(1), 1991.

[16] Maurice P. Herlihy and Jeannette M. Wing. Linearizability: A Correctness Condition for Concurrent Objects. *ACM Trans. Program. Lang. Syst.*, 12(3), 1990.

[17] Patrick Hunt, Mahadev Konar, Flavio P. Junqueira, and Benjamin Reed. Zookeeper: wait-free coordination for internet-scale systems. In *Proceedings of the 2010 USENIX conference on USENIX annual technical conference*, USENIX ATC'10, pages 11–11, Berkeley, CA, USA, 2010. USENIX Association.

[18] Flavio Paiva Junqueira, Ivan Kelly, and Benjamin Reed. Durability with bookkeeper. *Operating Systems Review*, 47(1):9–15, 2013.

[19] Rüdiger Kapitza, Johannes Behl, Christian Cachin, Tobias Distler, Simon Kuhnle, Seyed Vahid Mohammadi, Wolfgang Schröder-Preikschat, and Klaus Stengel. CheapBFT: resource-efficient Byzantine fault tolerance. In *EuroSys*, pages 295–308, 2012.

[20] M. Pease, R. Shostak, and L. Lamport. Reaching Agreement in the Presence of Faults. *J. ACM*, 27(2), 1980.

[21] Rodrigo Rodrigues and Barbara Liskov. High availability in DHTs: Erasure coding vs. replication. In *IPTPS*, pages 226–239, 2005.

[22] Giuliana Santos Veronese, Miguel Correia, Alysson Neves Bessani, Lau Cheuk Lung, and Paulo Veríssimo. Efficient byzantine fault-tolerance. *IEEE Trans. Computers*, 62(1):16–30, 2013.

[23] Werner Vogels. Eventually consistent. *Commun. ACM*, 52(1):40–44, 2009.

[24] Yang Wang, Lorenzo Alvisi, and Mike Dahlin. Gnothi: separating data and metadata for efficient and available storage replication. In *Proceedings of the 2012 USENIX conference on Annual Technical Conference*, USENIX ATC'12, pages 38–38, Berkeley, CA, USA, 2012. USENIX Association.

[25] Zhe Wu, Michael Butkiewicz, Dorian Perkins, Ethan Katz-Bassett, and Harsha V. Madhyastha. Spanstore: cost-effective geo-replicated storage spanning multiple cloud services. In *SOSP*, 2013.

[26] Jian Yin, Jean-Philippe Martin, Arun Venkataramani, Lorenzo Alvisi, and Michael Dahlin. Separating agreement from execution for byzantine fault tolerant services. In *SOSP*, pages 253–267, 2003.