Hybris: Robust Hybrid Cloud Storage

Dan Dobre
work done at NEC Labs Europe
dan@dobre.net

Paolo Viotti
EURECOM
paolo.viotti@eurecom.fr

Marko Vukolić
Systems Group, ETH Zurich and EURECOM
marko.vukolic@eurecom.fr

Abstract

Besides well-known benefits, commodity cloud storage also raises concerns that include security, reliability, and consistency. We present Hybris key-value store, the first *robust* hybrid cloud storage system, aiming at addressing these concerns leveraging both private and public cloud resources.

Hybris robustly replicates metadata on trusted private premises (private cloud), separately from data which is dispersed (using replication or erasure coding) across multiple untrusted public clouds. Hybris maintains metadata stored on private premises at the order of few dozens of bytes per key, avoiding the scalability bottleneck at the private cloud. In turn, the hybrid design allows Hybris to efficiently and robustly tolerate cloud outages, but also potential malice in clouds without overhead. Namely, to tolerate up to f malicious clouds, in the common case of the Hybris variant with data replication, writes replicate data across $f+1$ clouds, whereas reads involve a single cloud. In the worst case, only up to f additional clouds are used. This is considerably better than earlier multi-cloud storage systems that required costly $3f+1$ clouds to mask f potentially malicious clouds. Finally, Hybris leverages strong metadata consistency to guarantee to Hybris applications strong data consistency without any modifications to the eventually consistent public clouds.

We implemented Hybris in Java and evaluated it using a series of micro and macrobenchmarks. Our results show that Hybris significantly outperforms comparable multi-cloud storage systems and approaches the performance of bare-bone commodity public cloud storage.

Categories and Subject Descriptors C.2.4 [*Computer and Communication Networks*]: Distributed Systems

Keywords Cloud storage, Hybrid cloud, Reliability

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. For example, an enterprise can keep the sensitive data on premises while storing less sensitive data at potentially untrusted public clouds. In a sense, hybrid cloud eliminates to a large extent various security concerns that companies have with entrusting their data to commercial clouds [31] — as a result, enterprise-class hybrid cloud storage solutions are booming with all leading storage providers offering their proprietary solutions.

Beyond security and trust concerns, storing data to a single cloud poses issues related to provider reliability, availability, performance, vendor lock-in, as well as consistency, as cloud storage services are notorious for providing only eventual consistency [32]. To this end, several research works (e.g., [6–8, 10, 33, 35]) considered storing data *robustly* into public clouds, by leveraging *multiple* commodity cloud providers. In short, these *multi-cloud* storage systems, such as DepSky [8], ICStore [6], SPANStore [35] and SCFS [7], leverage multiple public cloud providers to distribute the trust across clouds, increase reliability, availability and consistency guarantees, and/or optimize the cost of using the cloud. A significant advantage of the multi-cloud approach is that it is based on client libraries that share data accessing commodity clouds, and as such, demands no big investments into proprietary storage solutions.

However, the existing robust multi-cloud storage systems suffer from serious limitations. Often, the robustness of these systems is limited to tolerating cloud outages, but not arbitrary or malicious behavior in clouds (e.g., data corruptions) — this is the case with ICStore [6] and SPANStore [35]. Other multi-cloud systems that do address malice in systems (e.g., DepSky [8] and SCFS [7]) require prohibitive cost of relying on $3f+1$ clouds to mask f faulty ones. This is a major overhead with respect to tolerating only cloud outages, which makes these systems expensive to use in practice. Moreover, all existing multi-cloud storage systems scatter storage metadata across public clouds increasing the difficulty of storage management and impacting performance.

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. By unifying the hybrid cloud with the multi-cloud, Hybris effectively brings together the best of both worlds, increasing security, reliability and consistency. At the same time, the novel design of Hybris allows for the first time to tolerate potentially malicious clouds at the price of tolerating only cloud outages.

Hybris exposes the de-facto standard key-value store API and is designed to seamlessly replace services such as Amazon S3 as the storage back-end of modern cloud applications. 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 approach not only allows more control over the data scattered around different public clouds, but also 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:

Tolerating cloud malice at the price of outages: 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 replicates data 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., [7, 8]).

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. Besides, by storing metadata locally on private cloud premises, Hybris avoids expensive round-trips for metadata operations that plagued previous multi-cloud storage systems. Finally, to reduce replication overhead Hybris optionally supports erasure coding [29], which reduces storage requirements at public clouds at the expense of increasing the number of clouds involved in reads/writes.

Scalability: The potential pitfall of using private cloud in combination with public clouds is in incurring a scalability bottleneck at a private cloud. Hybris avoids this pitfall by keeping the metadata very small. As an illustration, the replicated variant of Hybris maintains about 50 bytes of metadata per key, which is an order of magnitude less than comparable systems [8]. As a result, Hybris metadata service residing on a small commodity private cloud, can easily support up to 30k write ops/s and nearly 200k read ops/s, despite being fully replicated for metadata fault-tolerance.

Indeed, for Hybris to be truly robust, it has 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 [21]. 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 from ZooKeeper to SQL-based replicated RDBMS as well as NoSQL data stores that export conditional update operation (e.g., HBase or MongoDB).

Hybris offers per-key multi-writer multi-reader capabilities and guarantees linearizability (atomicity) [20] of reads and writes even in presence of eventually consistent public clouds [32]. To this end, Hybris relies on strong metadata consistency within a private cloud to mask potential inconsistencies at public clouds — in fact, Hybris treats cloud inconsistencies simply as arbitrary cloud faults. Our implementation of the Hybris metadata service over Apache ZooKeeper uses wait-free [19] concurrency control, boosting further the scalability of Hybris with respect to lock-based systems (e.g., SPANStore [35], DepSky [8] and SCFS [7]).

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. While different caching solutions can be applied to Hybris, we chose to interface Hybris with Memcached [2] distributed cache, deployed on the same machines that run ZooKeeper servers.

We implemented Hybris in Java[1] and evaluated it using both microbenchmarks and the YCSB [13] 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 Section 2, we present the Hybris architecture and system model. Then, in Section 3, we give the algorithmic aspects of the Hybris protocol. In Section 4 we discuss Hybris implementation and optimizations. In Section 5 we present Hybris performance evaluation. We overview related work in Section 6, and conclude in Section 7. Correctness arguments are postponed to Appendix A.

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)

[1] Hybris code is available at `http://hybris.eurecom.fr/code`.

storage resources and 2) (virtually unlimited) untrusted storage resources in public clouds. Hybris is designed to leverage commodity public cloud storage with API that does not offer computation, i.e., key-value stores (e.g., Amazon S3).

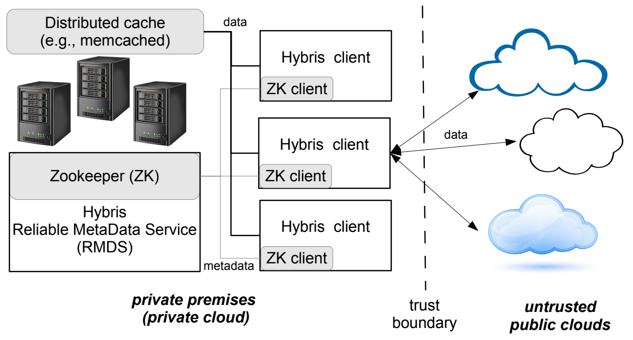


Figure 1. Hybris architecture. Reused (open-source) components are depicted in grey.

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 [2], 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. 3.6).

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

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 [27]) 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).[2] Namely, we assume that the communication among processes located in the private portion of the cloud is partially synchronous [16] (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 also 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 [32]; 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.

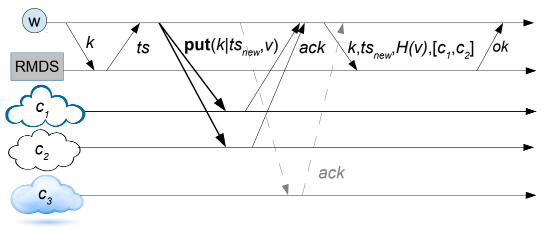
Finally, for simplicity, we assume an adversary that can coordinate malicious processes as well as process crashes. However, we assume that the adversary cannot subvert cryptographic hash functions we use (SHA-1), and that it cannot spoof the communication among non-malicious processes.

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; (iii) DELETE($cont, key$) to remove the respective entry and *(iv)* LIST($cont$) to list the keys present in container *cont*. We collectively refer to 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.

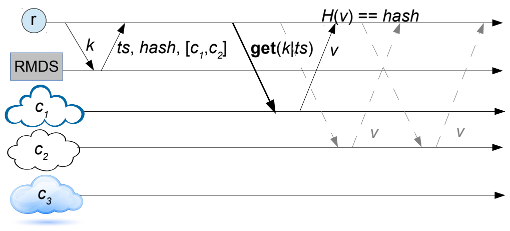
Hybris implements a multi-writer multi-reader key-value storage. Hybris is strongly consistent, i.e., it implements atomic (or *linearizable* [20]) semantics. In distributed storage context, atomicity provides an illusion that a complete operation *op* is executed instantly at some point in time between 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 are *wait-free*, i.e., writes by a correct client are guaranteed to eventually complete [19]. On the other hand, Hybris guarantees a read operation by a correct client to complete always, except in the corner case where an infinite number of writes to the same key is concurrent with the read operation (this is called finite-write termination [3]).

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



(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. Hybris Protocol

The key component of Hybris is its RMDS component which maintains metadata (logical timestamp, pointers to clouds storing data, cryptographic hash of data, and 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. 3.2). Additional f clouds (totaling $2f+1$ clouds) are only needed to guarantee that PUT is available as well (see Sec. 3.1).

Finally, besides cryptographic hash and pointers to clouds, metadata includes a timestamp that induces a partial order of operations which captures the real-time precedence ordering among operations (atomic consistency). The subtlety of Hybris (see Sec. 3.4 for details) is in the way it combines timestamp-based lock-free multi-writer concurrency control within RMDS with garbage collection (Sec. 3.3) of stale values from public clouds to save on storage costs.

In the following we detail each Hybris operation. Note that a given Hybris client never invokes concurrent operations on the same key; operations on the same key by the same client are invoked sequentially.

3.1 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. As this final step is the linearization point of PUT, it has to be performed in a specific way. 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.

3.2 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 and for that particular cloud). In the common case, the client is able to download the correct value from the first cloud c_1 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*.[3]

In specific corner cases, caused by concurrent garbage collection (described in Sec. 3.3), failures, repeated timeouts (asynchrony), or clouds' inconsistency, the client has to take additional actions in GET (described in Sec. 3.4).

3.3 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.

3.4 GET in the worst-case

In the context of cloud storage, there are known issues with weak, e.g., eventual [32] 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 3.2 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. 3.3).

To cope with these issues and eventual consistency in particular, Hybris leverages metadata service consistency to mask data inconsistencies in the clouds effectively allowing availability to be traded off for consistency. To this end, 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. 3.3), 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.., Apache ZooKeeper (using the concepts of *watches*, see Sec. 4 for details). The entire protocol executed only if common-case GET fails (Sec. 3.2) 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[4].
   (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 [3] 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. 3.3) might systematically and indefinitely often remove the written values before the client manages to retrieve them.[5]

3.5 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

[3] 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.

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

[5] 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.

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.

3.6 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. 3.1), 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}.

3.7 Erasure coding

In order to minimize bandwidth and storage capability requirements, Hybris supports erasure coding. Erasure codes entail partitioning data into $k > 1$ blocks with m additional parity blocks, each of the $k+m$ blocks taking about $1/k$ of the original storage space. When using an *optimal* erasure code, the data can be reconstructed from any k blocks despite up to m erasures. In Hybris, we fix m to equal f.

Deriving an erasure coding variant of Hybris from its replicated counterpart is relatively straightforward. Namely, in a PUT, the client encodes original data into $f+k$ erasure coded blocks and places one block per cloud. Hence, with erasure coding, PUT involves $f+k$ clouds in the common case (instead of $f+1$ with replication). Then, the client computes $f+k$ hashes (instead of a single hash with replication) that are stored in the RMDS as the part of metadata. Finally, erasure coded GET fetches blocks from k clouds in common case, with block hashes verified against those stored in RMDS. In the worst case, Hybris with erasure coding uses up to $2f+k$ (resp., $f+k$) clouds in PUT (resp., GET).

Finally, it is worth noting that in Hybris, there is no explicit relation between parameters f and k which are independent. This offers more flexibility with respect to prior solutions that mandated $k \geq f+1$.

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 about 3400 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 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 ZooKeeper API. Then, we describe several Hybris optimizations that we implemented.

4.1 ZooKeeper-based RMDS

We layered Hybris implementation over Apache ZooKeeper [21]. 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. 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**() are atomic.

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 [11]) is wait-free [19] and terminates as 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 **getData**($path_k$), just like in our PUT protocol. In addition, for subscriptions for metadata updates in GET (Sec. 3.4) 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 3.4.

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.

Erasure coding. Hybris integrates an optimally efficient Reed-Solomon codes implementation, using the Jerasure library [28], by means of its JNI bindings. The cloud latency rank optimizations remains in place with erasure coding. When performing a PUT, $f+k$ erasure coded blocks are stores in $f+k$ clouds with lowest latency, whereas with GET, $k>1$ clouds with lowest latency are selected (out of $f+k$ clouds storing data chunks).

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 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[6] 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.[7] We use the KVM hypervisor, and each machine in the physical cluster runs the Grizzly release of OpenStack on top of a Ubuntu 14.04 Linux distribution.

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 clients. Each VM has 100Mb/s internet connectivity for both upload and download bandwidths. Clients 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 $f=1$ [14], i.e., where at most one public cloud can exhibit arbitrary faults. Additionally, we fix the erasure coding reconstruction threshold k to 2.

Experiment 1: Common-case latency. In this experiment, we benchmark the common-case latency of Hybris and Hybris-EC (i.e. Hybris using erasure coding instead of replication) with respect to those of DepSky-A [8],[8] 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 10MB and output the median latencies together with 95% confidence intervals on boxplot graphs.

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

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

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

We repeated each experiment 30 times, and each set of GET and PUT operations has been performed one after the other in order to minimize 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.[9] 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). 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.

Hybris-EC PUT roughly performs as the third fastest cloud as it uploads 3 chunks in parallel, after having created the coding information. Similarly Hybris-EC GET performs slightly worse than the second fastest cloud because it retrieves chunks from 2 clouds in parallel before reconstructing the original data.

Notice that Hybris outperforms DepSky-A and DepSky-EC in both PUT and GET operations. The difference is significant particularly for smaller to medium object sizes (100kB and 1MB). This is explained by the fact that Hybris stores metadata locally, whereas DepSky needs to fetch metadata across clouds. With increased file sizes (10MB) network latency takes over and the difference is less pronounced.

Throughout the tests, we observed a significant variance in cloud performance and in particular for downloading large objects at Amazon and Rackspace.

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, by injecting a fault on the closest cloud, i.e. the one likely to be chosen for the download because of its low latency. We injected faults by manually tampering the data.

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: 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 6.

Figure 6(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.

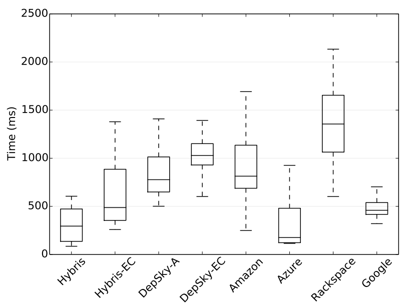
On the other hand, PUT operations incur the expense of atomic broadcast within ZooKeeper and stable storage accesses in the critical path. Figure 6(b) shows the latency-throughput curve for three different classes of stable storage backing ZooKeeper, namely conventional HDD, SSD and RAMDISK, which would 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 6(c) shows that despite 128 clients writing concurrently to the same key, the latency overhead is only 30% over clients writing to separate keys.

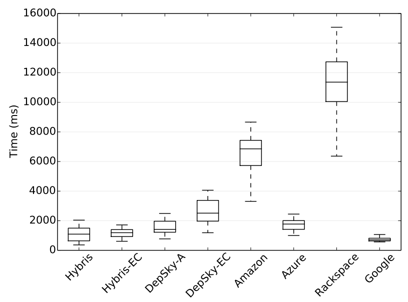
Finally, Figures 6(d) and 6(e) depict throughput curves as more clients performing operations in closed-loop are added to the system. Specifically, 6(d) suggests that ZooKeeper-based RMDS is able to service read requests coming from $2K$ clients near peak throughput. On the other hand, Figure 6(e) shows 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.

Experiment 4: Caching. In this experiment we test Hybris caching 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. We varied blob sizes from 1kB to 10 MB and measured 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 7.

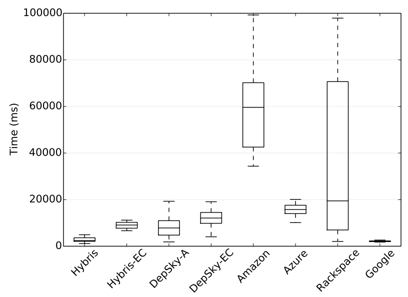
[9] In the boxplots the central line is showing the median, the box corresponds to 1st and 3rd quartiles while whiskers are drawn at the most extreme data points within 1.5 times the interquartile range from 1st and 3rd quartiles.



(a) 100kB GET

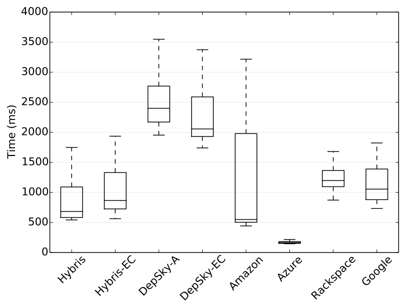


(b) 1MB GET

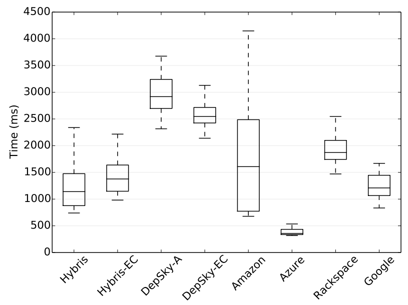


(c) 10MB GET

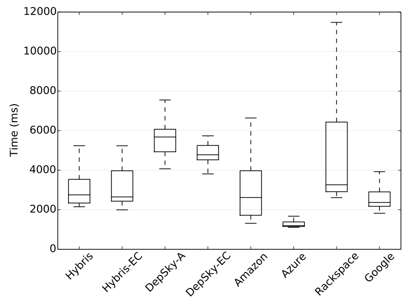
Figure 3. Latencies of GET operations.



(a) 100kB PUT

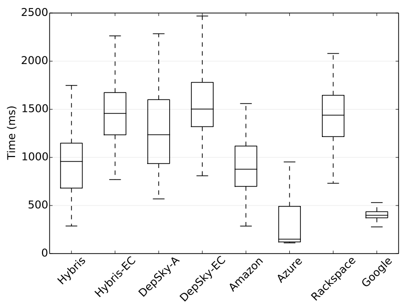


(b) 1MB PUT

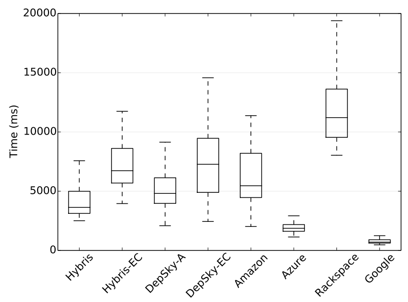


(c) 10MB PUT

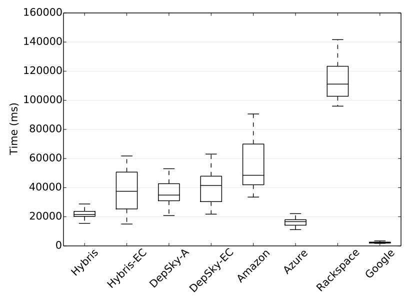
Figure 4. Latencies of PUT operations.



(a) 100kB GET



(b) 1MB GET



(c) 10MB GET

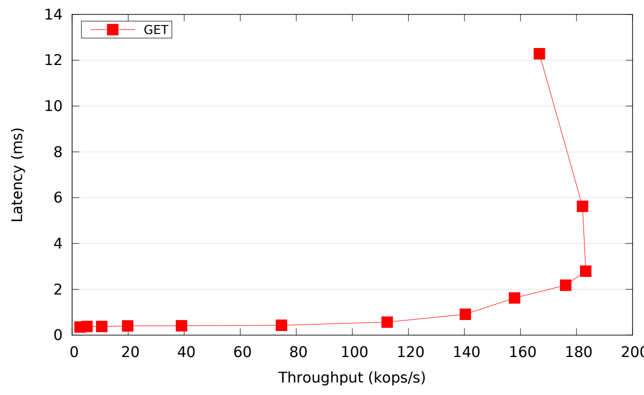
Figure 5. Latencies of GET operations with one faulty cloud.

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).

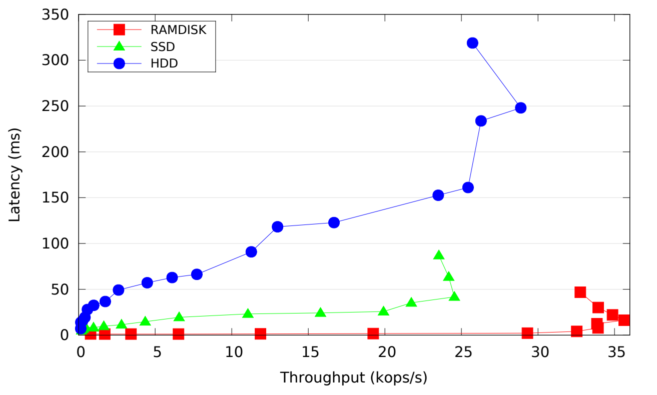
Cost comparison. Table 1 summarizes the monetary costs incurred by several cloud storage systems in the common case (i.e. synchrony, no failures, no concurrency), including Amazon S3 as the baseline. For the purpose of calculating costs, given in USD, we set $f = 1$ and assume a symmetric

System	PUT	GET	Storage Cost / Month	Total
ICStore [6]	60	376	144	580
DepSky-A [8]	30	376	72	478
DepSky-EC [8]	30	136	36	202
Hybris	10	120	48	178
Hybris-EC	15	120	36	171
Amazon S3	5	120	24	149

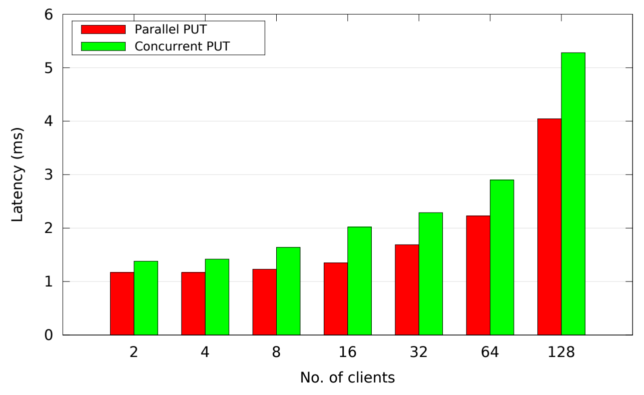
Table 1. Cost of cloud storage systems in USD for 2 x 10^6 transactions, and 10^6 files of 1MB, totaling to 1TB of storage.



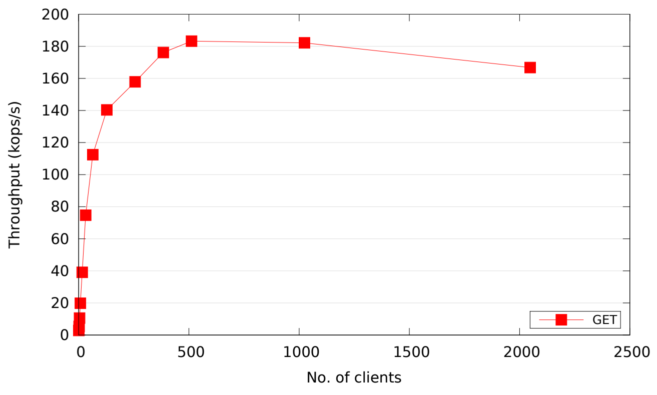
(a) GET latency



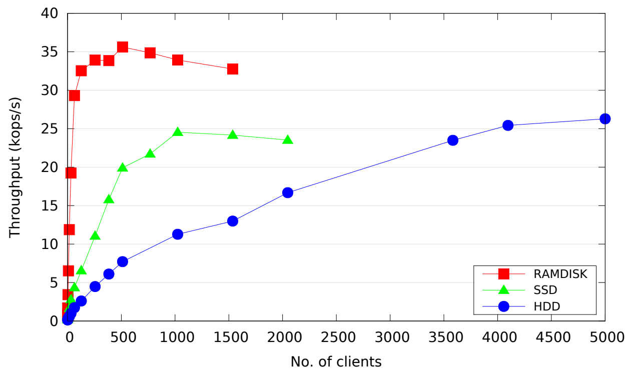
(b) PUT latency



(c) PUT latency under concurrency



(d) GET throughput



(e) PUT throughput

Figure 6. Performance of metadata read and write operations.

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. The cost per transaction, storage, and outbound traffic are taken from Amazon S3 as of 07/18/2014. The basis for costs calculation is Table 2. Our figures exclude the cost of private cloud infrastructure in Hybris, which we assume to be part of already existing IT infrastructure.

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. On the other hand Hybris-EC incurs for $k = 2$ a reduced storage overhead of $1.5x$ the baseline at the cost of increased overhead for PUT, because data needs to be dispersed onto 3 clouds.

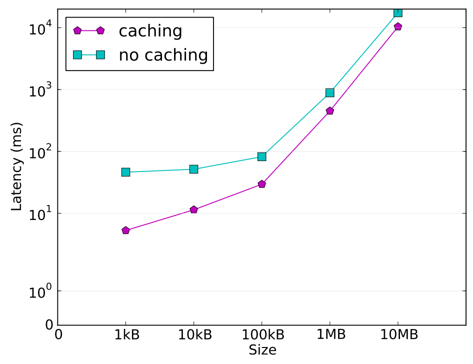


Figure 7. Hybris GET latency with YCSB workload B.

6. Related Work

Protocol	Semantics		Common case performance	
	Cloud faults	Consistency	No. of Cloud operations	Blow-up
ICStore [6]	crash-only	atomic[1]	$(4f+2)(D+m)$ (writes) $(2f+1)(D+m)$ (reads)	$4f+2$
DepSky [8]	**arbitrary**	regular[1]	$(2f+1)(D+m)$ (writes) $(2f+1)(D+m)$ (reads)	$2f+1$[2]
Hybris	**arbitrary**	**atomic**	$\mathbf{(f+1)D}$ (writes) $\mathbf{1D}$ (reads)	$\mathbf{f+1}$[3]

[1] Unlike Hybris, to achieve atomic (resp., regular) semantics, ICStore (resp., DepSky) require public clouds to be atomic (resp., regular).
[2] The erasure coded variant of DepSky features $\frac{2f+1}{f+1}$ storage blowup.
[3] The erasure coded variant of Hybris features $\frac{f+k}{k}$ storage blowup, for any $k > 1$.

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

Multi-cloud storage systems. Several storage systems (e.g., [4, 6, 8, 9, 24, 26, 35]) have used multiple clouds in boosting data robustness, notably reliability and availability. SafeStore [24] erasure codes data across multiple storage providers (clouds) providing data integrity, confidentiality and auditing, while using a centralized local proxy for concurrency control, caching and storing metadata. SPANStore [35] is another recent multi-cloud storage system that seeks to minimize the cost of use of multi-cloud storage, leveraging a centralized cloud placement manager. However, SafeStore and SPANStore are not robust in the Hybris sense, as their centralized components (local proxy and placement manager, respectively) are single point of failure. RACS [4] and HAIL [9] assume immutable data, hence not addressing any concurrency aspects. Depot [26] key-value store tolerates any number of untrusted clouds, yet does not offer strong consistency and requires computation on clouds.

Multi-cloud storage systems closest to Hybris are DepSky [8] and ICStore [6]. For clarity, we overview main aspects of these three systems in Table 2. ICStore models cloud faults as outages and implements robust access to shared data. Hybris advantages over ICStore include tolerating malicious clouds and smaller storage blowup[10]. 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 strongly consistent. Finally, Hybris guarantees atomicity even in presence of eventually consistent clouds, which may harm the consistency guarantees of both ICStore and DepSky. Recently, and concurrently with this work, SCFS [7] augmented DepSky to a full fledged file system by applying a similar idea of turning eventual consistency to strong consistency by separating cloud file system metadata from payload data that is stored using DepSky. Nevertheless, SCFS still requires $3f+1$ clouds to tolerate f malicious ones (the overhead SCFS inherits from DepSky).

Separating data from metadata. Separating metadata from data is not a novel idea in distributed systems. For example, Farsite [5] 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 [18], 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 [17] and BookKeeper [22]. LDR [17] 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 [22] 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. Unlike BookKeeper, Hybris supports multiple writers and tolerates malicious faults of data repositories.Interestingly, all robust crash-tolerant protocols that separate metadata from data (e.g., [17, 22], but also Gnothi [34]), 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 [36] in

[10] 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.

the context of replicated state machines (RSM). While the RSM approach of [36] 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 [36].

Systems based on trusted components. Several systems have used trusted hardware to reduce the overhead of replication despite malicious faults from $3f+1$ to $2f+1$ replicas, typically in the context of RSM (e.g., [12, 15, 23, 30]). Some of these systems, like CheapBFT [23], employ only $f+1$ replicas in the common case.

Conceptually, Hybris is similar to these systems in that it 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 entail placing trusted hardware 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 hybrid cloud model being in fact inspired by practical system deployments.

7. Conclusion and Future Work

In this paper we presented Hybris, the first robust hybrid storage system. Hybris disperses data (using replication or erasure coding) across multiple untrusted and possibly inconsistent public clouds, while it replicates metadata within trusted premises of a private cloud. Hybris tolerates up to f arbitrary public cloud faults and is very efficient: in the common-case (with replicated variant of Hybris), writes accesses only $f+1$ clouds, while a reads accesses a single, "closest" cloud. In a sense, Hybris is the first multi-cloud storage protocol that makes it possible to tolerate potentially malicious clouds at the price of tolerating simple cloud outages. To complement this, Hybris offers strong consistency as it leverages strong consistency of metadata stored off-clouds to mask the weak consistency of data stored in clouds.

Hybris is designed to seamlessly replace commodity key-value cloud storage (e.g., Amazon S3) in existing applications. Hybris could be used for archival ("cold") data, but also for mutable data due to its strong multi-writer consistency. For example, in future work and in the scope of the CloudSpaces EU project [1], we plan to integrate Hybris with a Dropbox-like personal cloud frontend [25].

In future work, we also plan to address the limitation of our current deployment of Hybris that restricts metadata servers to a single geographical location. We aim at geo-replicating Hybris metadata service, thereby accommodating geographically distributed clients.

Acknowledgments

We thank Phil Gibbons and anonymous SoCC reviewers for their useful comments and suggestions. We also thank Christian Cachin for his comments on earlier drafts of the paper. This work is partially supported by the EU projects CloudSpaces (FP7-317555) and SECCRIT (FP7-312758).

A. Correctness

Here, we provide the correctness arguments and pseudocode (Alg. 1) for the core part of the protocol (PUT, Sec. 3.1 and worst-case GET, Sec. 3.4). For space limitations, we omit the correctness proofs pertaining to the RMDS implementation and DELETE and LIST operations.[11] The atomic functionality of RMDS is however specified in Algorithm 2.

Algorithm 1 Algorithm of client *cid*.

```
1:  operation PUT $(k, v)$
2:      $(ts, -, -, -) \leftarrow RMDS.\text{READ}(k, false)$
3:      if $ts = \perp$ then $ts \leftarrow (0, cid)$
4:      $ts \leftarrow (ts.sn + 1, cid)$
5:      $cloudList \leftarrow \emptyset$; trigger$(timer)$
6:      forall $f+1$ selected clouds $C_i$ do
7:          put $\langle k|ts, v\rangle$ to $C_i$
8:      wait until $|cloudList| = f+1$ or $timer$ expires
9:      if $|cloudList| < f+1$ then
10:         forall $f$ secondary clouds $C_i$ do
11:             put $\langle k|ts, v\rangle$ to $C_i$
12:         wait until $|cloudList| = f+1$
13:     $RMDS.\text{CONDUPDATE}(k, ts, cloudList, H(v), size(v))$
14:     trigger garbage collection              // see Section 3.3
15:     return OK
16: upon put $\langle k|ts, v\rangle$ completes at cloud $C_i$
17:     $cloudList \leftarrow cloudList \cup \{i\}$
18: operation GET $(k)$            //worst-case, Section 3.4 code only
19:     $(ts, cloudList, hash, size) \leftarrow RMDS.\text{READ}(k, true)$
20:     if $ts = \perp$ then return $\perp$
21:     forall $C_i \in cloudList$ do
22:         get $(k|ts)$ from $C_i$
23: upon get $(k|ts)$ returns $data$ from cloud $C_i$
24:     if $H(data) = hash$ then return $data$
25:     else get $(k|ts)$ from $C_i$
26: upon received notify$(k, ts')$ from RMDS such that $ts' > ts$
27:     cancel all pending gets
28:     return GET $(k)$
```

Algorithm 2 RMDS functionality (atomic).

```
29: operation READ$(k, notify)$ by $cid$
30:     if $notify$ then $subscribed \leftarrow subscribed \cup \{cid\}$
31:     return $(ts(k), cList(k), hash(k), size(k))$
32: operation CONDUPDATE$(k, ts, cList, hash, size)$
33:     if $ts(k) = \perp$ or $ts > ts(k)$ then
34:         $(ts(k), cList(k), hash(k), size(k)) \leftarrow (ts, cList, hash, size)$
35:         send notify$(k, ts)$ to every $c \in subscribed(k)$
36:         $subscribed(k) \leftarrow \emptyset$
37:     return OK
```

[11] Moreover, we ignore possible DELETE operations — the proof is easy to modify to account for impact of DELETE.

We define the timestamp of operation o, denoted $ts(o)$, as follows. If o is a PUT, then $ts(o)$ is the value of client's variable ts when its assignment completes in line 4, Alg. 1. Else, if o is a GET, then $ts(o)$ equals the value of ts when client executes line 24, Alg. 1 (i.e., when GET returns). We further say that an operation o *precedes* operation o', if o completes before o' is invoked. Without loss of generality, we assume that all operations access the same key k.

LEMMA A.1 (Partial Order). *Let o and o' be two* GET *or* PUT *operations with timestamps $ts(o)$ and $ts(o')$, respectively, such that o precedes o'. Then $ts(o) \leq ts(o')$ and if o' is a* PUT *then $ts(o) < ts(o')$.*

Proof: In the following, prefix $o.$ denotes calls to RMDS within operation o (and similarly for o'). Let o' be a GET (resp. PUT) operation.
Case 1 (o is a PUT): then $o.RMDS.$CONDUPDATE$(o.md)$ in line 13, Alg. 1, precedes (all possible calls to) $o'.RMDS.$READ$()$ in line 19, Alg. 1 (resp., line 2, Alg. 1). By atomicity of RMDS (and RMDS functionality in Alg. 2) and definition of operation timestamps, it follows that $ts(o') \geq ts(o)$. Moreover, if o' is a PUT, then $ts(o') > ts(o)$ because $ts(o')$ is obtained from incrementing the timestamp ts returned by $o'.RMDS.$READ$()$ in line 2, Alg. 1, where $ts \geq ts(o)$.

Case 2 (o is a GET): then since (all possible calls to) $o'.RMDS.$READ$()$ in line 19 (resp. 2) follows after (the last call to) $o.RMDS.$READ$()$ in line 19, by Alg. 2 and atomicity of RMDS, it follows that $ts(o') \geq ts(o)$. If o' is a PUT, then $ts(o') > ts(o)$, similarly as in Case 1. □

LEMMA A.2 (Unique PUTs). *If o and o' are two* PUT *operations, then $ts(o) \neq ts(o')$.*

Proof: By lines 2- 4, Alg. 1, RMDS functionality (Alg. 2) and the fact that a given client does not invoke concurrent operations on the same key. □

LEMMA A.3 (Integrity). *Let rd be a* GET(k) *operation returning value $v \neq \bot$. Then there is a single* PUT *operation wr of the form* PUT(k,v) *such that $ts(rd) = ts(wr)$.*

Proof: Since rd returns v and has a timestamp $ts(rd)$, rd receives v in response to **get**$(k|ts(rd))$ from some cloud C_i. Suppose for the purpose of contradiction that v is never written by a PUT. Then, by the collision resistance of $H()$, the check in line 24 does not pass and rd does not return v. Therefore, we conclude that some operation wr issues **put** $(k|ts(rd))$ to C_i in line 7. Hence, $ts(wr) = ts(rd)$. Finally, by Lemma A.2 no other PUT has the same timestamp. □

THEOREM A.4 (Atomicity). *Every execution ex of Algorithm 1 satisfies atomicity.*

Proof: Let ex be an execution of the algorithm. By Lemma A.3 the timestamp of a GET either has been written by some PUT or the GET returns $\bot$. With this in mind, we first construct ex' from ex by completing all PUT operations of the form PUT (k,v), where v has been returned by some complete GET operation. Then we construct a sequential permutation π by ordering all operations in ex', except GET operations that return $\bot$, according to their timestamps and by placing all GET operations that did not return $\bot$ immediately after the PUT operation with the same timestamp. The GET operations that did return $\bot$ are placed in the beginning of π.

Towards atomicity, we show that a GET rd in π always returns the value v written by the latest preceding PUT which appears before it in π, or the initial value of the register $\bot$ if there is no such PUT. In the latter case, by construction rd is ordered before any PUT in π. Otherwise, $v \neq \bot$ and by Lemma A.3 there is a PUT (k,v) operation, with the same timestamp, $ts(rd)$. In this case, PUT (k,v) appears before rd in π, by construction. By Lemma A.2, other PUT operations in π have a different timestamp and hence appear in π either before PUT (k,v) or after rd.

It remains to show that π preserves real-time order. Consider two complete operations o and o' in ex' such that o precedes o'. By Lemma A.1, $ts(o') \geq ts(o)$. If $ts(o') > ts(o)$ then o' appears after o in π by construction. Otherwise $ts(o') = ts(o)$ and by Lemma A.1 it follows that o' is a GET. If o is a PUT, then o' appears after o since we placed each read after the PUT with the same timestamp. Otherwise, if o is a GET, then it appears before o' as in ex'. □

THEOREM A.5 (Availability). *Hybris* PUT *calls are wait-free, whereas Hybris* GET *calls are finite-write terminating.*

Proof: The wait freedom of Hybris PUT follows from: a) the assumption of $2f+1$ clouds out of which at most f may be faulty (and hence the **wait** statement in line 12, Alg. 1 is non-blocking), and b) wait-freedom of calls to RMDS (hence, calls to RMDS in lines 2 and 13, Alg. 1 return).

We prove finite-write termination of GET by contradiction. Assume there is a finite number of writes to key k in execution ex, yet that there is a GET(k) operation rd by a correct client that never completes. Let W be the set of all PUT operations in ex, and let wr be the PUT operation with maximum timestamp ts_{max} in W that completes the call to RMDS in line 13, Alg. 1. We distinguish two cases: (i) rd invokes an infinite number of recursive GET calls (in line 28, Alg 1), and (ii) rd never passes the check in line 24, Alg. 1.

In case (i), there is a recursive GET call in rd, invoked after wr completes conditional update to RMDS. In this GET call, the client does not execute line 28, Alg 1, by definition of wr and specification of $RMDS.$CONDUPDATE in Alg. 2 (as there is no **notify** for a $ts > ts_{max}$). A contradiction.

In case (ii), notice that key $k|ts_{max}$ is never garbage collected at $f+1$ clouds that constitute $cloudList$ in line 13, Alg. 1 in wr. Since rd does not terminate, it receives a notification in line 26, Alg. 1 with timestamp ts_{max} and reiterates GET. In this iteration of GET, the timestamp of rd is ts_{max}. As $cloudList$ contains $f+1$ clouds, including at least one correct cloud C_i, and as C_i is eventually consistent, C_i eventually returns value v written by wr to a **get** call. This value v passes the hash check in line 24, Alg. 1 and rd completes. A contradiction. □

References

[1] CloudSpaces EU FP7 project. `http://cloudspaces.eu/`.

[2] Memcached. `http://memcached.org/`.

[3] I. Abraham, G. Chockler, I. Keidar, and D. Malkhi. Byzantine Disk Paxos: Optimal Resilience with Byzantine Shared Memory. *Distributed Computing*, 18(5):387–408, 2006.

[4] H. Abu-Libdeh, L. Princehouse, and H. Weatherspoon. RACS: a case for cloud storage diversity. In *SoCC*, pages 229–240, 2010.

[5] A. Adya, W. J. Bolosky, M. Castro, G. Cermak, R. Chaiken, J. R. Douceur, J. Howell, J. R. Lorch, M. Theimer, and R. P. Wattenhofer. Farsite: Federated, available, and reliable storage for an incompletely trusted environment. *SIGOPS Oper. Syst. Rev.*, 36(SI), Dec. 2002.

[6] C. Basescu, C. Cachin, I. Eyal, R. Haas, A. Sorniotti, M. Vukolić, and I. Zachevsky. Robust data sharing with key-value stores. In *Proceedings of DSN*, pages 1–12, 2012.

[7] A. Bessani, R. Mendes, T. Oliveira, N. Neves, M. Correia, M. Pasin, and P. Verissimo. SCFS: A shared cloud-backed file system. In *Usenix ATC*, 2014.

[8] A. N. Bessani, M. Correia, B. Quaresma, F. André, and P. Sousa. DepSky: Dependable and secure storage in a cloud-of-clouds. *ACM Transactions on Storage*, 9(4):12, 2013.

[9] K. D. Bowers, A. Juels, and A. Oprea. HAIL: a high-availability and integrity layer for cloud storage. In *ACM CCS*, pages 187–198, 2009.

[10] C. Cachin, R. Haas, and M. Vukolić. Dependable storage in the intercloud. Technical Report RZ3783, IBM Research, 2010.

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

[12] B.-G. Chun, P. Maniatis, S. Shenker, and J. Kubiatowicz. Attested append-only memory: making adversaries stick to their word. In *SOSP*, pages 189–204, 2007.

[13] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears. Benchmarking cloud serving systems with ycsb. In *SoCC*, pages 143–154, 2010.

[14] J. C. Corbett, J. Dean, M. Epstein, A. Fikes, C. Frost, J. J. Furman, S. Ghemawat, A. Gubarev, C. Heiser, P. Hochschild, W. C. Hsieh, S. Kanthak, E. Kogan, H. Li, A. Lloyd, S. Melnik, D. Mwaura, D. Nagle, S. Quinlan, R. Rao, L. Rolig, Y. Saito, M. Szymaniak, C. Taylor, R. Wang, and D. Woodford. Spanner: Google's globally distributed database. *ACM Trans. Comput. Syst.*, 31(3):8, 2013.

[15] M. Correia, N. F. Neves, and P. Veríssimo. How to tolerate half less one Byzantine nodes in practical distributed systems. In *SRDS*, pages 174–183, 2004.

[16] C. Dwork, N. Lynch, and L. Stockmeyer. Consensus in the presence of partial synchrony. *J. ACM*, 35(2):288–323, Apr. 1988. ISSN 0004-5411. .

[17] R. Fan and N. Lynch. Efficient Replication of Large Data Objects. In *Proceedings of DISC*, pages 75–91, 2003.

[18] S. Ghemawat, H. Gobioff, and S.-T. Leung. The google file system. In *SOSP*, pages 29–43, 2003.

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

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

[21] P. Hunt, M. Konar, F. P. Junqueira, and B. Reed. Zookeeper: wait-free coordination for internet-scale systems. In *USENIX ATC'10*, pages 11–11, 2010.

[22] F. P. Junqueira, I. Kelly, and B. Reed. Durability with bookkeeper. *Operating Systems Review*, 47(1):9–15, 2013.

[23] R. Kapitza, J. Behl, C. Cachin, T. Distler, S. Kuhnle, S. V. Mohammadi, W. Schröder-Preikschat, and K. Stengel. CheapBFT: resource-efficient Byzantine fault tolerance. In *EuroSys*, pages 295–308, 2012.

[24] R. Kotla, L. Alvisi, and M. Dahlin. Safestore: A durable and practical storage system. In *USENIX ATC*, pages 129–142, 2007.

[25] P. G. Lopez, S. Toda, C. Cotes, M. Sanchez-Artigas, and J. Lenton. Stacksync: Bringing elasticity to dropbox-like file synchronization. In *ACM/IFIP/USENIX Middleware*, 2014.

[26] P. Mahajan, S. T. V. Setty, S. Lee, A. Clement, L. Alvisi, M. Dahlin, and M. Walfish. Depot: Cloud storage with minimal trust. *ACM Trans. Comput. Syst.*, 29(4):12, 2011.

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

[28] J. S. Plank, S. Simmerman, and C. D. Schuman. Jerasure: A library in C/C++ facilitating erasure coding for storage applications - Version 1.2. Technical Report CS-08-627, University of Tennessee, August 2008.

[29] R. Rodrigues and B. Liskov. High availability in dhts: Erasure coding vs. replication. In *IPTPS*, pages 226–239, 2005.

[30] G. S. Veronese, M. Correia, A. N. Bessani, L. C. Lung, and P. Veríssimo. Efficient byzantine fault-tolerance. *IEEE Trans. Computers*, 62(1):16–30, 2013.

[31] VMware Professional Services. The Snowden Leak: A Windfall for Hybrid Cloud? `http://blogs.vmware.com/consulting/2013/09/the-snowden-leak-a-windfall-for-hybrid-cloud.html`.

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

[33] M. Vukolić. The Byzantine empire in the intercloud. *SIGACT News*, 41(3):105–111, 2010.

[34] Y. Wang, L. Alvisi, and M. Dahlin. Gnothi: separating data and metadata for efficient and available storage replication. In *USENIX ATC'12*, pages 38–38, 2012.

[35] Z. Wu, M. Butkiewicz, D. Perkins, E. Katz-Bassett, and H. V. Madhyastha. Spanstore: cost-effective geo-replicated storage spanning multiple cloud services. In *SOSP*, 2013.

[36] J. Yin, J.-P. Martin, A. Venkataramani, L. Alvisi, and M. Dahlin. Separating agreement from execution for byzantine fault tolerant services. In *SOSP*, pages 253–267, 2003.