Hybris: Robust Hybrid Cloud Storage

Paolo Viotti, Dan Dobre, Marko Vukolić

A

Hybris: Robust Hybrid Cloud Storage

Paolo Viotti, Sorbonne Universités, UPMC, LIP6 [1]
Dan Dobre, work done at NEC Labs Europe
Marko Vukolić, IBM Research - Zurich

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 macro-benchmarks. 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: []

CCS Concepts: •**Information systems** → **Cloud based storage;** •**Hardware** → **Fault tolerance;** •**Software and its engineering** → **Consistency;**

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

Additional Key Words and Phrases: Cloud storage, hybrid cloud, reliability, consistency

ACM Reference Format:
Paolo Viotti and Dan Dobre and Marko Vukolić. 2016. Hybris: Robust Hybrid Cloud Storage. *ACM Trans. Storage* V, N, Article A (YYYY), 33 pages.
DOI: http://dx.doi.org/10.1145/0000000.0000000

1. INTRODUCTION

Hybrid cloud storage entails storing data on private premises as well as on one (or more) remote, public cloud storage platforms. 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), in addition to a fine-grained control over confidential data. For example, an enterprise can keep private data on premises

[1] work done at Eurecom.

A preliminary version of this article appeared in *Proceedings of the 5th ACM Symposium on Cloud Computing (SoCC 2014)*.
This work is partially supported by the EU projects CloudSpaces (FP7-317555) and SECCRIT (FP7-312758).

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

while storing less sensitive data at potentially untrusted public clouds. In a sense, the hybrid cloud approach eliminates to a large extent various security concerns that companies have with entrusting their data to commercial clouds [VMware 2013]. As a result, enterprise-class hybrid cloud storage solutions are booming, with all leading storage providers, such as Dell EMC,[2] IBM,[3] NetApp,[4] Microsoft[5] and others, offering proprietary solutions.

Besides security and trust concerns, storing data on a single cloud presents issues related to reliability [Gunawi et al. 2016], performance, vendor lock-in [Armbrust et al. 2010; Abu-Libdeh et al. 2010], as well as consistency, since cloud storage services are notorious for typically providing only eventual consistency [Vogels 2009; Bermbach and Tai 2014]. To address these concerns, several research works considered storing data *robustly* in public clouds, by leveraging *multiple* cloud providers [Armbrust et al. 2010; Vukolić 2010]. In short, these *multi-cloud* storage systems, such as DepSky [Bessani et al. 2013], ICStore [Basescu et al. 2012], SPANStore [Wu et al. 2013] and SCFS [Bessani et al. 2014], leverage multiple public cloud providers to distribute trust, increase reliability, availability and consistency guarantees. A significant advantage of the multi-cloud approach is that it is implemented on the client side and as such, it demands no big investments into additional storage solutions.

However, 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) [Basescu et al. 2012; Wu et al. 2013]. Other multi-cloud systems that do address arbitrary faults [Bessani et al. 2013; Bessani et al. 2014] require prohibitive costs, as they rely on $3f + 1$ clouds to mask f faulty ones. This is a significant 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 metadata across public clouds, increasing the difficulty of storage management, and impacting performance and costs.

In this paper, we unify the hybrid and the multi-cloud approaches, and present Hybris,[6] the first robust hybrid cloud storage system. By combining the hybrid cloud with the multi-cloud, Hybris effectively brings together the benefits of both paradigms, thereby increasing security, reliability and consistency. Additionally, the novel design of Hybris allows to withstand arbitrary cloud faults at the same price of tolerating only outages.

Hybris exposes the de facto standard key-value store API, and is designed to seamlessly replace popular storage services such as Amazon S3 as backend of modern cloud applications. The key idea behind Hybris is to keep the *metadata* on private premises, including metadata related to data outsourced to public clouds. This approach not only grants more control over the data scattered across different public clouds, but also allows Hybris to significantly outperform existing multi-cloud storage systems, both in terms of system performance (e.g., latency) and storage cost, while providing strong consistency guarantees.

In summary, the salient features of Hybris are the following:

[2]https://www.emc.com/en-us/cloud/hybrid-cloud-computing/index.htm.
[3]https://www.ibm.com/cloud-computing/bluemix/hybrid.
[4]http://www.netapp.com/us/solutions/cloud/hybrid-cloud/index.aspx.
[5]https://www.microsoft.com/en-us/cloud-platform/hybrid-cloud.
[6]Hybris, sometimes also transliterated from ancient Greek as 'hubris', means extreme pride or arrogance. In Greek mythology, Hybris describes heroic mortals striving to surpass the boundaries of their mortal nature, and/or defy the authority of gods.

Tolerating cloud malice at the price of outages. Hybris puts no trust in any public cloud provider. Namely, Hybris can mask arbitrary (including malicious) faults of up to f public clouds by replicating data on as few as $f + 1$ clouds in the common case (when the system is synchronous and without faults). In the worst case, that is, to cope with network partitions, cloud inconsistencies and faults, Hybris uses up to f additional clouds. This is in sharp contrast with existing multi-cloud storage systems that require up to $3f + 1$ clouds to mask f malicious ones [Bessani et al. 2013; Bessani et al. 2014]. Additionally, Hybris uses symmetric-key encryption to preserve the confidentiality of outsourced data. The required cryptographic keys are stored on trusted premises and shared through the metadata service.

Efficiency. Hybris is efficient and incurs low cost. In the common case, a Hybris write involves as few as $f + 1$ public clouds, whereas a read involves only a single cloud, even though all clouds are untrusted. Hybris achieves this using cryptographic hashes, and without relying on expensive cryptographic primitives. By storing metadata on local premises, Hybris avoids the expensive round-trips for lightweight operations that plagued previous multi-cloud systems. Finally, Hybris optionally reduces storage requirements by supporting erasure coding [Rodrigues and Liskov 2005], at the expense of increasing the number of clouds involved.

Scalability. The potential pitfall of adopting such compound architecture is that private resources may represent a scalability bottleneck. Hybris avoids this issue by keeping the metadata footprint very small. As an illustration, the replicated variant of Hybris maintains about 50 bytes of metadata per key, which is an order of magnitude smaller than comparable systems [Bessani et al. 2013]. As a result, the 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 fault tolerance. Moreover, Hybris offers per-key multi-writer multi-reader capabilities thanks to wait-free [Herlihy 1991] concurrency control, further boosting the scalability of Hybris compared to lock-based systems [Wu et al. 2013; Bessani et al. 2013; Bessani et al. 2014].

Strong consistency. Hybris guarantees linearizability (i.e. atomic consistency) [Herlihy and Wing 1990] of reads and writes even though public clouds may guarantee no more than eventual consistency [Vogels 2009; Bermbach and Tai 2014]. Weak consistency is an artifact of the high availability requirements of cloud platforms [Gilbert and Lynch 2002; Brewer 2012], and is often cited as a major impediment to cloud adoption, since eventually consistent stores are notoriously difficult to program and reason about [Bailis and Ghodsi 2013]. Even though some cloud stores have recently started offering strongly consistent APIs, this offer usually comes with significantly higher monetary costs (for instance, Amazon charges twice the price for strong consistency compared to weak [Amazon 2016]). In contrast, Hybris is cost-effective as it relies on strongly consistent metadata within a private cloud, which is sufficient to mask inconsistencies of the public clouds. In fact, Hybris treats a cloud inconsistency simply as an arbitrary fault. In this regard, Hybris implements one of the few known ways of composing consistency semantics in a practical and meaningful fashion.

We implemented Hybris as a Java application library.[7] To maintain its code base small and facilitate adoption, we chose to reliably replicate metadata by layering Hy-

[7]The Hybris prototype is released as open source software [Eurecom 2016].

bris on top of the Apache ZooKeeper coordination service [Hunt et al. 2010]. Hybris clients act simply as ZooKeeper clients — our system does not entail any modifications to ZooKeeper, hence easing its deployment. In addition, we designed Hybris metadata service to be easily portable from ZooKeeper to any SQL-based replicated RDBMS as well as NoSQL data store that exports a conditional update operation. As an example, we implemented alternative metadata services based on the Consul coordination service [Consul 2016b] and the XPaxos cross-fault tolerant metadata store [Liu et al. 2016]. We evaluated Hybris using both micro-benchmarks and the YCSB [Cooper et al. 2010] benchmarking framework. 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 guarantees. Furthermore, our scalability benchmarks attest that Hybris is an appealing solution for data storage of small and medium enterprises.

The rest of this article is organized as follows. In Section 2, we present the Hybris architecture and system model. Then, in Section 3, we provide the algorithmic details of the Hybris protocol. In Section 4, we discuss Hybris implementation and optimizations, whose performance we evaluate in Section 5. We overview related work in Section 6, and conclude in Section 7. Pseudocode of algorithms and correctness arguments are postponed to Appendix A.

2. HYBRIS OVERVIEW

The high-level design of Hybris is presented in Figure 1. Hybris mixes two types of resources: 1) private, trusted resources that provide computation and limited storage capabilities and 2) virtually unlimited untrusted storage resources in public clouds. We designed Hybris to leverage commodity cloud storage APIs that do not offer computation services, e.g., key-value stores like Amazon S3.

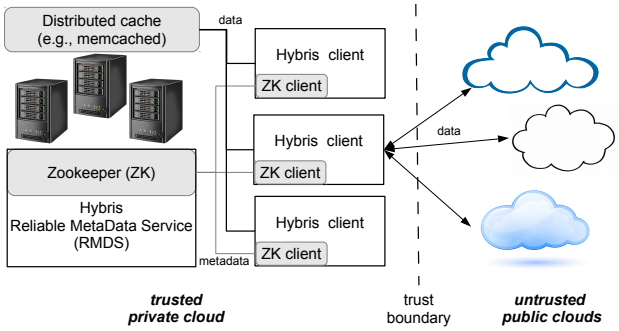


Fig. 1. Hybris architecture. Reused (open-source) components are depicted in grey.

Hybris stores data and metadata separately. Metadata is stored within the key component of Hybris called Reliable MetaData Service (RMDS). RMDS has no single point of failure and is assumed to reside on private premises.[8]

On the other hand, data is stored on untrusted public clouds. Hybris distributes data across multiple cloud storage providers for robustness, i.e., to mask cloud outages and even malicious faults. In addition to storing data on public clouds, Hybris supports data caching on private premises. While different caching solutions exist,

[8]We discuss and evaluate the deployment of RMDS across geographically distributed (yet trusted) data centers in Section 5.4.

our reference implementation reuses Memcached [Memcached 2016], an open source distributed caching system.

Finally, at the heart of the system is the Hybris client, whose library orchestrates the interactions with public clouds, RMDS and the caching service. The Hybris client is also responsible for encrypting and decrypting data, leveraging RMDS in order to share encryption keys (see Sec. 3.8).

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

2.1. System model

Fault model. We assume a distributed system where any of the components might fail. In particular, we consider a dual fault model, where: (i) the processes on private premises (i.e., in the private cloud) can fail by crashing,[9] and (ii) we model public clouds as prone to arbitrary failures, including malicious faults [Pease et al. 1980]. 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, to guarantee availability despite up to f (arbitrary) faulty public clouds, Hybris requires at least $2f + 1$ public clouds in total. However, Hybris consistency (i.e., *safety*) is maintained regardless of the number of faulty public clouds.

For simplicity, we assume an adversary that can coordinate malicious processes as well as process crashes. However, the adversary cannot subvert the cryptographic hash (e.g., SHA-2), and it cannot spoof communication among non-malicious processes.

Timing assumptions. Similarly to our fault model, our communication model is dual, with its boundary coinciding with the trust boundary (see Fig. 1). Namely, we assume communication within the private portion of the system as partially synchronous [Dwork et al. 1988] (i.e. with arbitrary but finite periods of asynchrony), whereas communication between 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.

We believe that our dual fault and timing assumptions reasonably reflect typical hybrid cloud deployment scenarios. In particular, the accuracy of this model finds confirmations in recent studies about performance and faults of public clouds [Gunawi et al. 2016] and on-premise clusters [Cano et al. 2016].

Consistency. 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 stores [Bermbach and Tai 2014]. 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 [Vogels 2009].

2.2. Hybris data model and semantics

Similarly to commodity public cloud storage services, Hybris exposes a key-value store (KVS) API. In particular, the Hybris address space consists of flat containers, each holding multiple keys. The KVS API consists of four main operations: *(i)* $\text{PUT}(cont, key, value)$, to put $value$ under key in container $cont$; *(ii)* $\text{GET}(cont, key)$, to retrieve the value associated with key; *(iii)* $\text{DELETE}(cont, key)$ to remove the key en-

[9] We relax this assumption by discussing the suitability of the *cross fault tolerance* (XFT) model [Liu et al. 2016] in §4.3. In §5.3 we evaluate the performance of both crash fault and cross fault tolerant replication protocols.

try and *(iv)* LIST($cont$) to list the keys present in container $cont$. Moreover, Hybris supports transactional writes through the TPUT($cont, \langle key_{lst} \rangle, \langle value_{lst} \rangle$) API. We collectively refer to operations that modify storage state (e.g., PUT, TPUT 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, and it is strongly consistent, i.e., it implements *linearizable* [Herlihy and Wing 1990] semantics. In the distributed storage context, linearizability (also known as atomic consistency) provides the illusion that the effect of a complete operation op takes place instantly at some point in time between its invocation and response. An operation invoked by a faulty client might appear either as complete or not invoked at all. Optionally, Hybris can be set to support weaker consistency semantics, which may enable better performance (see Sec. 3.10).

Although it provides strong consistency, Hybris is highly available. Hybris writes are *wait-free*, i.e., writes by a correct client are guaranteed to eventually complete [Herlihy 1991]. On the other hand, a Hybris read operation by a correct client will always complete, 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* [Abraham et al. 2006]). Hence, in Hybris, we trade read wait-freedom for finite-write termination and better performance. In fact, guaranteeing read wait-freedom reveals very costly in KVS-based multi-cloud storage systems [Basescu et al. 2012] and significantly impacts storage complexity. We feel that our choice will not be limiting in practice, since FW-termination essentially offers the same guarantees as wait-freedom for a large number of workloads.

3. HYBRIS PROTOCOL

In this section we present the Hybris protocol. We describe in detail how data and metadata are accessed by clients in the common case, and how consistency and availability are preserved despite failures, asynchrony and concurrency. We postpone the correctness proofs to Appendix A.

3.1. Overview

The key part of Hybris is the Reliable MetaData Store (RMDS), which maintains metadata associated with each key-value pair. Each metadata entry consists of the following elements: *(i)* a logical timestamp, *(ii)* a list of at least $f+1$ pointers to clouds that store value v, *(iii)* a cryptographic hash of v ($H(v)$), and *(iv)* the size of value v.

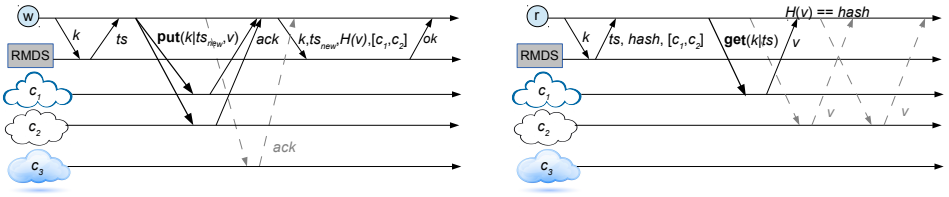
Despite being lightweight, the metadata is powerful enough to allow tolerating arbitrary cloud failures. Intuitively, the cryptographic hash within a trusted and consistent RMDS enables end-to-end integrity protection: neither corrupted nor stale data produced by malicious or inconsistent clouds are ever returned to the application. Additionally, the data size entry helps preventing certain denial-of-service attack vectors by a malicious cloud (see Sec. 4.4).

Furthermore, Hybris metadata acts as a directory pointing to $f+1$ clouds, thus enabling a client to retrieve the correct value despite f of them being arbitrarily faulty. In fact, with Hybris, as few as $f+1$ clouds are sufficient to ensure both consistency and availability of read operations (namely GET; see Sec. 3.3). Additional f clouds (totaling $2f+1$ clouds) are only needed to guarantee that writes (i.e. PUT) are available as well in the presence of f cloud outages (see Sec. 3.2).

Finally, besides cryptographic hash and pointers to clouds, each metadata entry includes a timestamp that induces a total order on operations which captures their real-time precedence ordering, as required by linearizability. Timestamps are managed by the Hybris client, and consist of a classical multi-writer tag [Lynch and Shvartsman

2002] comprising a monotonically increasing sequence number sn and a client id cid serving as tiebreaker.[10] The subtlety of Hybris lies 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 (see Sec. 3.5 for details).

In the following we detail each Hybris operation. We assume that a given Hybris client never invokes multiple concurrent operations on the same key.



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

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

3.2. PUT protocol

Hybris PUT protocol consists of the 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. Based on timestamp $ts = (sn, cid_i)$, 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 object size. In the common case, the $f + 1$ clouds reply to the client in a timely fashion, that is, 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 can proceed 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 timestamp ts_{new}, 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. This final step constitutes the *linearization point* of PUT, therefore it has to be performed in a specific way. Namely, if the client performs a straightforward update of metadata in RMDS, then this metadata might be overwritten by metadata with a *lower* timestamp (i.e. the so-called *old-new inversion* happens), breaking the timestamp ordering of operations and thus, violating linearizability.[11] In order to prevent this, we require RMDS to export an atomic conditional update operation. Hence, in the final step of Hybris PUT, the client issues a conditional update to RMDS, which updates the metadata for key k *only if* the written timestamp ts_{new} is *greater* than the one that RMDS already stores. In Section 4 we describe how we implemented this functionality over Apache ZooKeeper API and, alternatively, in the Consul-based

[10] We decided against leveraging server-managed timestamps (e.g., provided by ZooKeeper) to avoid constraining RMDS to a specific implementation. More details about RMDS implementations can be found in Sec. 4.

[11] Note that, since garbage collection (detailed in Sec. 3.4) relies on timestamp-based ordering to tell old values from new ones, old-new inversions could even lead to data loss.

RMDS instance. We note that any other NoSQL and SQL DBMS that supports conditional updates can be adopted to implement the RMDS functionality.

3.3. GET in the common case

The 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 for k, consisting 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. 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. The client concurrently starts a timer set to the typically observed download latency from c_1 (given the value size s). In the common case, the client is able to download the value v from the first cloud c_1 before expiration of its timer. Once it receives value v, the client checks that v matches the hash h included in the metadata bundle (i.e. if $H(v) = h$). If the value passes this check, then the client returns it 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 another cloud from $cloudList$ (see dashed arrows in Fig. 2(b)) and so on, until it exhausts all $f+1$ clouds from $cloudList$.[12] In some corner cases, caused by concurrent garbage collection (described in Sec. 3.4), failures, repeated timeouts (asynchrony), or clouds' inconsistency, the client must take additional actions, which we describe in Sec. 3.5.

3.4. Garbage collection

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

In the context of cloud storage, there are known issues with weak (e.g., eventual [Vogels 2009]) consistency. With eventual consistency, even a correct, non-malicious cloud might deviate from linearizable semantics and return an unexpected value, typically a stale one. In this case, the sequential common-case reading from $f+1$ clouds as described in Section 3.3 might not return the correct value, since the hash verification might fail at all $f+1$ clouds. In addition to the case of inconsistent clouds, this anomaly might also occur if: *(i)* the timers set by the client for otherwise non-faulty clouds expire (i.e. in case of asynchrony or network outages), and/or *(ii)* the values read by the client were concurrently garbage collected (see Sec. 3.4).

To address this issues, Hybris levereges strong metadata consistency to mask data inconsistencies in the clouds, effectively allowing availability to be traded off for consistency. To this end, the Hybris client indulgently reissues a *get* to all clouds in parallel, and waits to receive at least one value matching the required hash. However, due

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

to possible concurrent garbage collection (Sec. 3.4), the client needs to make sure it always compares the values received from clouds to the most recent key's metadata. This can be achieved in two ways: *(i)* by simply iterating over the entire GET including metadata retrieval from RMDS, or *(ii)* by only repeating the *get* operations at $f + 1$ clouds while fetching metadata from RMDS only when it actually changes.

In Hybris, we adopt the latter approach. Notice that this implies 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 by using, e.g., Apache ZooKeeper and Consul (through the concept of *watch*, see Sec. 4 for details). This worst-case protocol is executed only if the common-case GET fails (Sec. 3.3), and it proceeds as follows:

(1) The client first reads the metadata for key k from RMDS (i.e. timestamp ts, hash h, size s and cloud list $cloudList$) and subscribes for updates related to key k metadata.
(2) The client issues a parallel *get*$(k|ts)$ to 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.[13]
   (a) If the hash verification succeeds, the GET returns v_c.
   (b) Otherwise, the client discards v_c and reissues *get*$(k|ts)$ to cloud c.
(*) At any point in time, if the client receives a metadata update notification for key k from RMDS, it cancels all pending downloads, and repeats the procedure from step 1.

The complete Hybris GET, as described above, ensures finite-write termination [Abraham et al. 2006] in presence of eventually consistent clouds. Namely, a GET may fail to return a value only theoretically, i.e. in case of an infinite number of concurrent writes to the same key, in which case, garbage collection might systematically and indefinitely often remove every written value before the client manages to retrieve it.[14] We believe that this exceptional corner case is of marginal importance for the vast majority of applications.

3.6. Transactional PUT

Hybris supports a transactional PUT operation that writes atomically to multiple keys. The steps associated with the transactional PUT operation are depicted in Figure 3.

Similarly to the normal PUT, the client first fetches the latest authoritative timestamps $[ts_0...ts_n]$ by issuing parallel requests to the RMDS for metadata of the concerned keys $[k_0...k_n]$. Each timestamp ts_i is a tuple consisting of a sequence number sn_i and a client id cid_i. Based on timestamp ts_i, the client computes a new timestamp ts_{i_new} for each key, whose value is $(sn_i + 1, cid_i)$. Next, the client combines each key k_i and timestamp ts_{i_new} to a new key $k_{i\_new} = k_i|ts_{i\_new}$ and invokes *put* (k_{i_new}, v_i) on $f+1$ clouds in parallel. This operation is executed in parallel for each key to be written. Concurrently, the client starts a set of timers as for the normal PUT. In the common case, the $f + 1$ clouds reply to the client for each key in a timely fashion, before the timer expires. Otherwise, the client invokes *put* (k_{i_new}, v_i) to up to f secondary clouds. Once the client has received acknowledgments from $f + 1$ different clouds for each key, it is assured that the transactional PUT is durable and can thus proceed to the final stage of the operation.

In the final step, the client stores in RMDS the updated metadata associated with each key k_i, consisting of the timestamp ts_{i_new}, the cryptographic hash $H(v_i)$, and

[13] For simplicity, we model the absence of a value as a special NULL value that can be hashed.
[14] 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 list of pointers to the $f + 1$ clouds that have correctly stored v_i. As for the normal PUT operation, to avoid the so-called old-new inversion anomaly, we employ the conditional update exposed by RMDS. The metadata update succeeds only if, for each key k_i the written timestamp ts_{i_new} is greater than the timestamp currently stored for key k_i. In order to implement transactional atomicity, we wrap the metadata updates into an RMDS transaction. Specifically, we employ the MULTI API exposed by Apache ZooKeeper and the corresponding API in Consul. Thanks to this, if any of the single write to RMDS fails, the whole transactional PUT aborts. In this case, the objects written to the cloud stores are eventually erased by the normal garbage collection background task.

In summary, this approach implements an optimistic transactional concurrency control that, in line with the other parts of Hybris protocol, eschews locks to provide wait-freedom [Herlihy 1991].

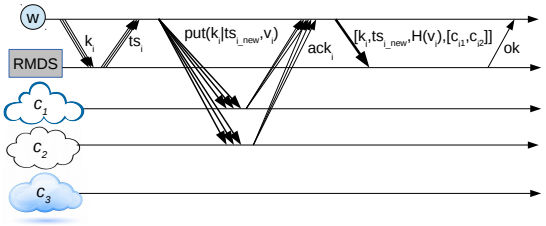


Fig. 3. Hybris transactional PUT protocol ($f = 1$). Worst case communication patterns are omitted for clarity.

3.7. DELETE and LIST

The Hybris DELETE and LIST operations are local to RMDS, and do not access public clouds.

In order to delete a value, the client performs the PUT protocol with the special $cloudList$ value $\perp$ denoting the deletion. Deleting a value creates a metadata *tombstone* in RMDS, i.e. metadata that lack corresponding values in the cloud stores. Metadata tombstones are necessary to keep record of the latest authoritative timestamp associated with a given key, and to preserve per-key timestamp monotonicity. Deleted values are eventually removed from cloud stores by the normal garbage collection. On the other hand, the LIST operation simply retrieves from RMDS all the keys in the container $cont$ that are not associated with tombstone metadata.

3.8. Confidentiality

Ensuring data confidentiality[15] in Hybris is straightforward. During a PUT, just before uploading data to $f + 1$ public clouds, the client encrypts the data with a symmetric cryptographic key k_{enc} which is then added to the metadata bundle. The hash is then computed on the ciphertext (rather than plaintext). The rest of PUT protocol remains unchanged. Notice that the client may generate a new encryption key at each PUT, or reuse the key stored in RMDS by previous PUT operations.

[15]Oblivious RAM algorithms can provide further confidentiality guarantees by masking data access patterns [Stefanov et al. 2013]. However, we decided not to integrate those algorithms in Hybris since they require performing additional operations and using further storage space, which could hinder performance and significantly increase monetary costs.

In order to decrypt data, a client uses the encryption key k_{enc} retrieved with the metadata bundle. Then, as the ciphertext downloaded from some cloud successfully passes the hash test, the client decrypts the data using k_{enc}.

3.9. Erasure coding

In the interest of minimizing bandwidth and storage space requirements, Hybris supports erasure coding. Erasure codes have been shown to provide resilience to failures through redundancy schemes which are significantly more efficient than replication [Weatherspoon and Kubiatowicz 2002]. Erasure codes entail partitioning data into $k > 1$ blocks with m additional parity blocks. Each of the $k + m$ blocks takes approximately $1/k$ of the original storage space. If the erasure code is *information-optimal*, the data can be reconstructed from any k blocks despite up to m erasures. In the context of cloud storage, blocks can be stored on different clouds and erasures correspond to arbitrary failures (e.g., network outages, data corruption, etc.). For simplicity, 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 operation, the client encodes original data into $f + k$ erasure-coded blocks, and stores 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 as with replication) that are stored in the RMDS as part of the metadata. Finally, the erasure-coded GET fetches blocks from k clouds in the 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) operations.

Finally, it is worth noting that in Hybris the parameters f and k are independent. This offers more flexibility with respect to prior solutions which mandated $k \geq f + 1$.

3.10. Weaker consistency semantics

A number of today's cloud applications may benefit from improved performance in exchange for weaker consistency guarantees. Over the years, researchers and practitioners have defined these weaker consistency guarantees in a wide spectrum of semantics [Viotti and Vukolic 2016]. Hybris exposes this consistency vs performance tradeoff to the application developers through an optional API. Specifically, Hybris implements two weaker consistency semantics: *read-my-write* and *bounded staleness* consistency.

Read-your-write. In read-my-writes consistency [Terry et al. 1994] a read operation invoked by some client can be serviced only by replicas that have already applied all previous write operations by the same client. E-commerce shopping carts are typical examples of applications that would benefit from this consistency semantics. Indeed, customers only write and read their own cart object, and are generally sensitive to the latency of their operations [Hamilton 2009].

This semantics is implemented in Hybris by leveraging caching. Essentially, a *write-through* caching policy is enabled in order to cache all the data written by each client. After a successful PUT, a client stores the written data in Memcached, under the compound key used for the clouds (i.e. $\langle k|ts_{new}\rangle$, see Sec. 3.2). Additionally, the client stores the compound key in a local in-memory hash table along with the original one (i.e. k). Later reads will fetch the data from the cache using the compound key cached locally. In this way, clients may obtaining previously written values without incurring the monetary and performance costs entailed by strongly consistent reads. In case of a cache miss, the client falls back to a normal read from the clouds as discussed in Sec. 3.3 and 3.5.

Bounded staleness. According to the bounded staleness semantics, the data read from a storage system must be fresher than a certain threshold. This threshold can be defined in terms of data versions [Golab et al. 2011], or real-time [Torres-Rojas et al. 1999]. Web search applications are a typical use case of this semantics, as they are latency-sensitive, yet they tolerate a certain bounded inconsistency.

Our bounded staleness protocol also makes use of the cache layer. In particular, to implement time-based bounded staleness we cache the written object on Memcached under the original key k — instead of using, as for read-your-write, the compound key. Additionally, we instruct the caching layer to evict all objects older than a certain expiration period Δ.[16] Hence, all objects read from cache will abide the staleness restriction.

To implement version-based bounded staleness, we add a counter field to the metadata stored on RMDS, accounting for the number of versions written since the last caching operation. During a PUT, the client fetches the metadata from RMDS (as specified in Sec. 3.2) and reads this caching counter. In case of successful writes to the clouds, the client increments the counter. If the counter exceeds a predefined threshold η, the object is cached under its original key (i.e. k) and the counter is reset. When reading, clients will first try to read the value from the cache, thus obtaining, in the worst case, a value that is η versions older than the most recent one.

4. IMPLEMENTATION

We implemented Hybris as an application library [Eurecom 2016]. The implementation pertains solely to the Hybris client side since the entire functionality of the metadata service (RMDS) is layered on top of the Apache ZooKeeper client. Namely, Hybris does not entail any modification to the ZooKeeper server side. Our Hybris client is lightweight and consists of about 3800 lines of Java code. Hybris client interactions with public clouds are implemented by wrapping individual native Java SDK clients (drivers) for each cloud storage provider into a common lightweight interface that masks the small differences across the various storage APIs.[17]

In the following, we first discuss in detail our RMDS implementation with ZooKeeper and the alternative one using Consul; then we describe several Hybris optimizations that we implemented.

4.1. ZooKeeper-based RMDS

We layered our reference Hybris implementation over Apache ZooKeeper [Hunt et al. 2010]. In particular, we durably store Hybris metadata as ZooKeeper *znodes*. In ZooKeeper, znodes are data objects addressed by *paths* in a hierarchical namespace. 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 offers a fairly modest API. The ZooKeeper API calls relevant to Hybris are the following:

— *create*/*setData*($p, data$) creates/updates a znode identified by path p with $data$.
— *getData*(p) is used to retrieve data stored under znode p.
— *sync()* synchronizes the ZooKeeper replica that maintains the client's session with the ZooKeeper leader, thus making sure that the read data contains the latest updates.

[16] Similarly to Memcached, most modern off-the-shelf caching systems implement this functionality.
[17] Initially, our implementation relied on the Apache JClouds library [Apache 2016], which roughly serves the main purpose as our custom wrappers, yet covers dozens of cloud providers. However, JClouds introduces its own performance overhead that prompted us to implement the cloud driver library wrapper ourselves.

— *getChildren(p)* (only used in Hybris LIST) returns the list of znodes whose paths are prefixed by p.

Finally, ZooKeeper allows several operations to be wrapped into a transaction, which is then executed atomically. We used this API to implement the TPUT (transactional PUT) operation.

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

ZooKeeper linearizable reads. In ZooKeeper, only write operations are linearizable [Hunt et al. 2010]. In order to get the latest updates through the *getData* calls, the recommended technique consists in performing a *sync* operation beforehand. While this normally results in a linearizable read, there exists a corner case scenario in which another quorum member takes over as leader, while the old leader, unaware of the new configuration due to a network partition, still services read operations with possibly stale data. In such case, the read data would still reflect the update order of the various clients but may fail to include recent completed updates. Hence, the "*sync+read*" schema would result in a *sequentially consistent* read [Lamport 1979]. This scenario would only occur in presence of network partitions (which are arguably rare on private premises), and in practice it is effectively avoided through the use of heartbeats and timeouts mechanisms between replicas [Hunt et al. 2010]. Nonetheless, in principle, the correctness of a distributed algorithm should not depend on timing assumptions. Therefore we implemented an alternative, *linearizable* read operation through the use of a dummy write preceding the actual read. This dummy write, being a normal quorum-based operation, synchronizes the state among replicas and ensures that the following read operation reflects the latest updates seen by the current leader. With this approach, we trade performance for a stronger consistency semantics (i.e., linearizability [Herlihy and Wing 1990]). We implemented this scheme as an alternative set of API calls for the ZooKeeper-based RMDS, and benchmarked it in a geo-replicated setting (see Sec. 5.4) — as it represents the typical scenario in which this kind of tradeoffs are most conspicuous. However, for simplicity of presentation, in the following we only refer to the *sync+read* schema for getting data from the ZooKeeper-based RMDS.

Hybris PUT. At the beginning of PUT(k, v), when the client fetches the latest timestamp ts for k, the Hybris client issues a *sync*() followed by *getData*($path_k$). This *getData* call returns, besides Hybris timestamp ts, the internal version number vn of the znode $path_k$. In the final step of PUT, the client issues *setData*($path_k, md, vn$) which succeeds only if the version of znode $path_k$ is still vn. If the ZooKeeper version of $path_k$ has 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 [Chockler et al. 2013]) is wait-free [Herlihy 1991], thus always terminates, as only a finite number of concurrent PUT operations use a timestamp smaller than ts.

Hybris GET. During GET, the Hybris client reads metadata from RMDS in a strongly consistent fashion. To this end, a client always issues a *sync*() followed by *getData*($path_k$), just like in the PUT protocol. In addition, to subscribe for metadata up-

dates in GET we use ZooKeeper *watches* (set by, e.g., *getData* calls). In particular, we make use of these notifications in the algorithm described in Section 3.5.

4.2. Consul-based RMDS

In order to further study Hybris performance, we implemented an alternative version of RMDS using Consul [Consul 2016b]. Like ZooKeeper, Consul is a distributed coordination service, which exposes a simple key-value API to store data addressed in a URL-like fashion. Consul is written in Go and implements the Raft consensus algorithm [Ongaro and Ousterhout 2014]. Unlike ZooKeeper, Consul offers a service discovery functionality and has been designed to support cross-data center deployments.[18]

The implementation of the Consul RMDS client is straightforward, as it closely mimics the logic described in Sec. 4.1 for ZooKeeper. Among the few relevant differences we note that the Consul client is stateless and uses HTTP rather than a binary protocol. Furthermore, Consul reads can be linearizable without the need for additional client operations to synchronize replicas.

4.3. Cross fault tolerant RMDS

The recent widespread adoption of portable connected devices has blurred the ideal security boundary between trusted and untrusted settings. Additionally, partial failures due to misconfigurations, software bugs and hardware failures in trusted premises have a record of causing major outages in productions systems [Correia et al. 2012]. Recent research by Ganesan et al. [2017] has highlighted how in real-world crash fault tolerant stores even minimal data corruptions can go undetected or cause disastrous cluster-wide effects. For all these reasons, it is arguably sensible to adopt replication protocols robust enough to tolerate faults beyond crashes even in trusted premises. Byzantine fault tolerant (BFT) replication protocols are an attractive solution for dealing with these issues. However, BFT protocols are designed to handle failure modes which are unreasonable for systems running in trusted premises, as they assume active and even malicious adversaries. Besides, handling such powerful adversaries takes a high toll on performance. Hence, several recent research works have proposed fault models that stand somewhere in-between the crash and the Byzantine fault models. A prominent example of this line of research is *cross fault* tolerance (XFT) [Liu et al. 2016], which decouples faults due to network disruptions from arbitrary machine faults. Basically, this model excludes the possibility of an adversary that controls both the network and the faulty machines at the same time. Thus, it fittingly applies to systems deployed in private premises. Therefore, we implemented an instance of RMDS that guarantees cross fault tolerance. We omit implementation details because, as the server side code is based on ZooKeeper [Liu et al. 2016], the client side logic closely mimics the one implemented for the ZooKeeper-based RMDS.

4.4. Optimizations

Cloud latency ranking. In our Hybris implementation, clients rank clouds by latency and prioritize those that present lower latency. Hybris client then uses these cloud latency rankings 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* to retrieve objects in GET. Initially, we implemented the cloud latency ranking by reading once (i.e. upon initialization of the Hybris client) a default, fixed-size (e.g., 100kB) object from each of the public clouds. Interestingly, during our experiments, we observed

[18] Currently, the recommended way of deploying Consul across data centers is by using separate consensus instances through partitioning of application data (see https://www.consul.io/docs/internals/consensus.html).

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 rankings depending on the file size and the type of operation. In addition, the 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 [Plank et al. 2008], by means of its JNI bindings. The cloud latency ranking optimization 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 might mount a DoS attack against an Hybris client during a read by sending, instead of the correct object, an object of arbitrary large size. In this way, a client would not detect a malicious fault until computing a hash of the received file. To cope with this attack, the Hybris client saves object size s as metadata on RMDS and cancels the downloads whose payload length exceeds s.

Caching. Our Hybris implementation enables object caching on private portions of the system. We implemented simple write-through cache and caching-on-read policies. With write-through caching enabled, the Hybris client simply writes to cache in parallel to writing to the clouds. On the other hand, with caching-on-read enabled, the Hybris client asynchronously writes the GET object to cache, upon returning it to the application. In our implementation, we use Memcached distributed cache, which exports a key-value API 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, after fetching metadata from RMDS, always tries first to read data from the cache (i.e., by issuing $get(k|ts)$ to Memcached). Only in case of a cache miss, it proceeds normally with a GET, as described in Sections 3.3 and 3.5.

Furthermore, Hybris can be instructed to use the caching layer to provide specific consistency semantics weaker than linearizability, as described in Sec. 3.10.

5. EVALUATION

In this section we evaluate Hybris performance, costs and scalability in various settings. In detail, we present the following experiments:

(1) An evaluation of common-case latency of Hybris compared to a a state-of-the-art multi-cloud storage system [Bessani et al. 2013], as well as to the latency of individual cloud providers (5.1).
(2) An evaluation of the GET latency with one malicious fault in a public cloud (5.2).
(3) A scalability benchmark of the Hybris RMDS component in its crash fault and cross fault tolerant implementations (5.3).
(4) A benchmark of RMDS scalability in a wide area deployment (5.4).
(5) An evaluation of Hybris caching performance using YCSB cloud serving benchmark [Cooper et al. 2010] (5.5).
(6) An assessment of Hybris as backend of a personal storage and synchronization application (5.6).
(7) An estimate of the monetary costs of Hybris compared to alternatives (5.7).

In all the following experiments, unless specified otherwise, caching is disabled. We focus on the arguably most common and interesting case where $f = 1$ [Corbett et al. 2013], i.e., where at most one public cloud may exhibit arbitrary faults. Furthermore,

we set the erasure coding reconstruction threshold k to 2. Hybris clients interact with four cloud providers: Amazon S3, Rackspace CloudFiles, Microsoft Azure and Google Cloud Storage. For each provider, we only used cloud storage data centers located in Europe.

5.1. 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, DepSky-EC (i.e., a version of DepSky featuring erasure codes support) [Bessani et al. 2013],[19] and the four individual public clouds underlying both Hybris and DepSky.

Private cloud setup. To perform this experiment and the next one (Sec. 5.2), we deployed Hybris "private" components (namely, Hybris client, metadata service (RMDS) and cache) on virtual machines (VMs) within an OpenStack[20] 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 RAID5 hard-drive volumes and two 1Gb/s network interfaces; nine worker nodes that execute on two sixteen-core Intel Xeon CPU E5-2630 servers clocked at 2.4GHz, with 128GB of RAM, ten 1TB disks and four 1Gb/s network cards.[21] We use the KVM hypervisor, and each machine in the physical cluster runs the Juno release of OpenStack on top of a Ubuntu 14.04 Linux distribution.

We collocate ZooKeeper and Memcached (in their off-the-shelf default configurations) using three VMs. Each VM has one quad-core virtual processor clocked at 2.40GHz, 8GB 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 16.04 distribution images, updated with the most recent patches. In addition, several OpenStack VMs with similar features are used for running clients. Each VM has 100Mb/s internet connectivity for both upload and download bandwidths.

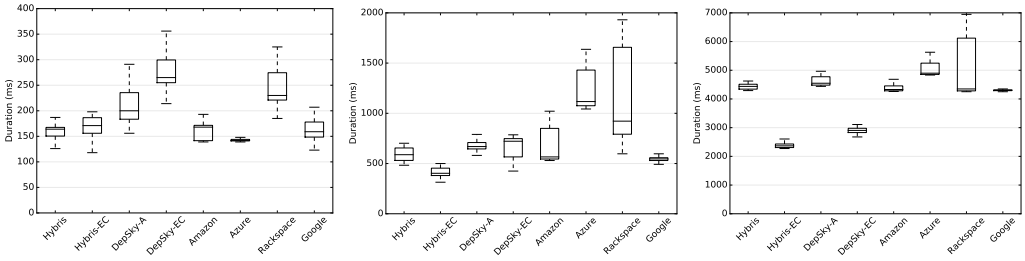
For this micro-benchmark we perform a set of isolated PUT and GET operations for data sizes ranging from 100kB to 10MB stored under a single key. We repeated each experiment 30 times, and each set of GET and PUT operations has been performed one after the other in order to minimize side effects due to internet routing and traffic fluctuations.

Figures 4 and 5 show the boxplots of client latencies, varying the size of the object to be written or read. In the boxplots, the central line shows the median, the box corresponds to the 1^{st} and 3^{rd} quartiles, and whiskers are drawn at the most extreme data points within 1.5 times the interquartile range from 1^{st} and 3^{rd} quartiles.

We observe that Hybris GET latency (Fig. 4) closely follows those of the fastest cloud storage provider, as in fact it downloads the object from that specific cloud, thanks to Hybris cloud latency ranking (see Sec. 4). We further observe (Fig. 5) that Hybris PUT 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.

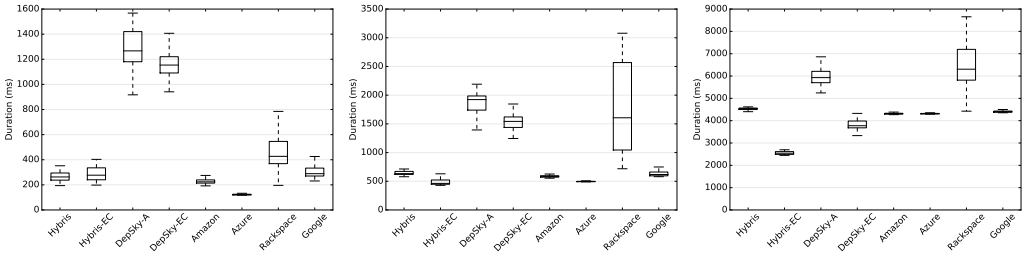
Hybris-EC PUT uploads 3 chunks roughly half as large as the original payload, in parallel, to the three fastest clouds. Notice that the overhead of computing the coding information and of using a third cloud is amortized as the payload size increases.

[19] We used the open-source DepSky implementation available at http://cloud-of-clouds.github.io/depsky/.
[20] http://www.openstack.org/
[21] Our hardware and network configuration closely resembles the one recommended by commercial private cloud providers.



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

Fig. 4. Latencies of GET operations.



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

Fig. 5. Latencies of PUT operations.

Similarly, Hybris-EC GET retrieves chunks of about half the original size from the two fastest clouds in parallel. As for the PUT, Hybris-EC GET performance advantage increases as the payload size increases.

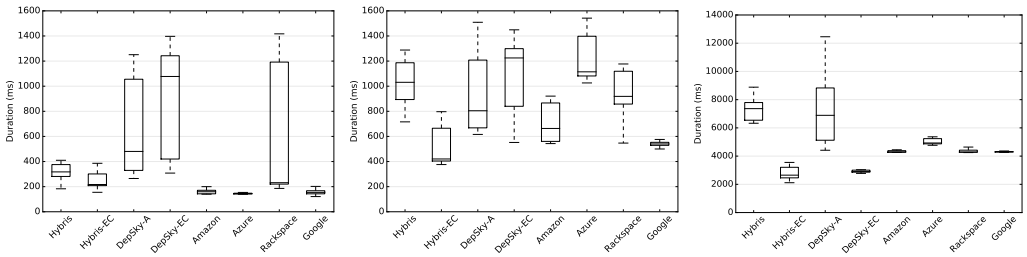
Notice that Hybris and Hybris-EC outperform the corresponding clients of DepSky in both PUT and GET operations. The difference is significant particularly for smaller to medium object sizes (e.g., 100kB and 1MB). This is explained by the fact that Hybris stores metadata locally, whereas DepSky needs to store and fetch metadata across clouds. With increased file sizes (e.g., 10MB) latency merely due to payload takes over and the difference becomes less pronounced.

We further note that we observed, throughout the tests, a significant variance of clouds performance, in particular for downloading large objects from Amazon and Rackspace. However, thanks to its latency ranking, Hybris manages to mitigate the backlashes of this phenomenon on the overall performance.

5.2. Experiment 2: latency under faults

In order to assess the impact of faulty clouds on Hybris GET performance, we repeated 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 most likely to be chosen for the download because of its low latency. We injected faults by manually tampering the data through an external client.

Figure 6 shows the download times of Hybris, Hybris-EC, 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



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

Fig. 6. Latencies of GET operations with one faulty cloud.

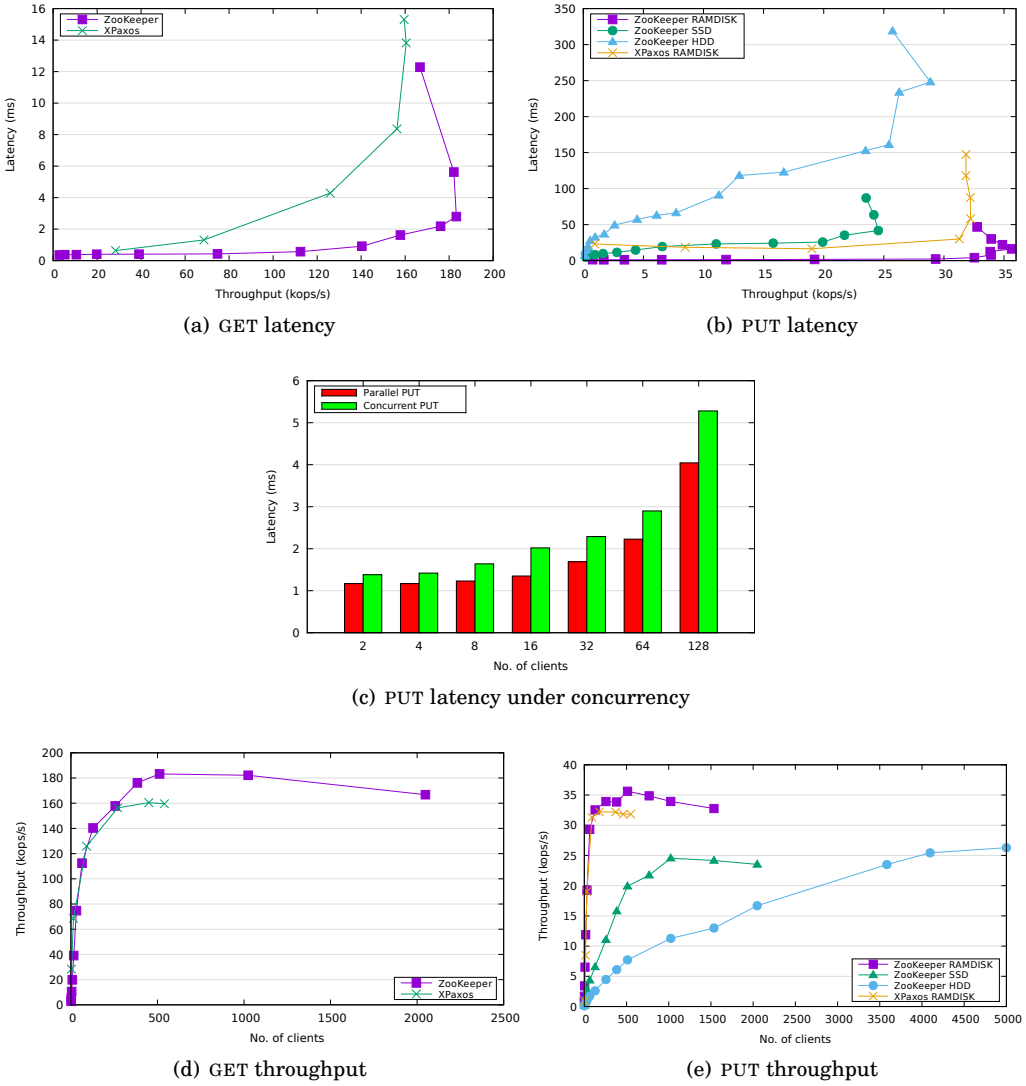
GET downloads happen, in this case, sequentially. However, despite its single cloud read optimization, Hybris performance under faults remains comparable to that of DepSky variants that download objects in parallel. We further note that both Hybris-EC and DepSky-EC are less sensitive to faulty clouds than the corresponding versions featuring plain replication, as they fetch fewer data in parallel from single clouds.

5.3. Experiment 3: RMDS performance

As we envision a typical deployment of Hybris in corporate settings, which generally present high Internet access bandwidth, we identify in RMDS the most likely bottleneck of the system. Therefore, in this experiment we aim to stress our crash fault tolerant (vanilla ZooKeeper) and cross fault tolerant (XPaxos [Liu et al. 2016]) RMDS implementations in order to assess their performance. For this purpose, we short-circuit public clouds and simulate uploads by writing a 100 byte payload to an in-memory hash map. To mitigate possible performance impact of the shared OpenStack private cloud, we perform (only) this experiment deploying RMDS on a dedicated cluster of three 8-core Xeon E3-1230 V2 machines (3.30GHz, 20GB ECC RAM, 1GB Ethernet, 128GB SATA SSD, 250GB SATA HDD 10000rpm). The obtained results 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 come with a modest overhead, and we take advantage of read locality in ZooKeeper to balance requests across nodes. Furthermore, since RMDS has a small footprint, all read requests are serviced directly from memory without incurring the cost of stable storage access. Using the XPaxos-based RMDS, Hybris GET achieves a peak of 160 kops/s with latencies of about 10 ms. For read operations, XPaxos message pattern is similar to ZooKeeper's and it uses lightweight cryptographic operations (e.g., message authentication codes).

In contrast, PUT operations incur the toll of atomic broadcast 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. XPaxos achieves a maximum throughput of about 32 kops/s using RAMDISK as storage. The difference between ZooKeeper and XPaxos performance is due to the use of CPU-intensive cryptographic



(a) GET latency

(b) PUT latency

(c) PUT latency under concurrency

(d) GET throughput

(e) PUT throughput

Fig. 7. Performance of metadata read and write operations with RMDS deployed as local cluster in private premises.

operations in XPaxos. Note that, unlike in [Liu et al. 2016], XPaxos does not outperform ZooKeeper because, in the cluster setting of a private cloud, CPU is the bottleneck of XPaxos, whereas in the WAN experiments by Liu et al. [2016] the bottleneck, for both protocols, is the network. Nevertheless, the peak throughput of XPaxos-based RMDS is within 10% of ZooKeeper peak throughput, which seems an acceptable overhead for the additional guarantees of XPaxos and the XFT model.[22]

[22]An optimized implementation of XPaxos could achieve better performance by offloading cryptographic operations to a dedicated cryptoprocessor.

To understand the impact of concurrency on RMDS performance, we evaluated the latency of PUT under heavy contention to a single key.[23] Figure 7(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 7(d) and 7(e) depict throughput curves as more clients invoking operations in closed-loop are added to the system. Specifically, Fig. 7(d) suggests that ZooKeeper-based RMDS is able to service read requests coming from $2K$ clients near peak throughput, while XPaxos can service up to 600 clients on the same client machines due to its substantial use of cryptographic operations. On the other hand, Figure 7(e) shows again the performance discrepancy in ZooKeeper 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.

While clearly these scalability results do not match the needs of very large companies, we believe that Hybris would serve well small and medium enterprises — while significantly improving over the consistency and reliability guarantees offered by comparable single-cloud and private-cloud systems.

5.4. Experiment 4: RMDS geo-replication

Modern applications are being deployed increasingly often across wide area networks, in so-called *geo-replicated* settings [Corbett et al. 2013], to improve latency and/or fault tolerance. To evaluate Hybris performance in this context, we placed each of the servers composing the RMDS cluster in a different data center, and replicated the measurements of Sec. 5.3.

For this experiment, we used virtual machines and network infrastracture by IBM SoftLayer [SoftLayer 2016]. Specifically, three virtual machines make up the RMDS cluster, while three others host processes emulating concurrent clients invoking GET and PUT operations. We placed two machines (one for the clients and one as a server) in each of three data centers located in San Jose (US), Washington D.C. (US) and London (UK). All the machines run Ubuntu 16.04 and dispose of 4GB RAM, one quad-core 2.6GHz CPU, a 24GB SSD virtual hard-drive, and 100Mbps public VLAN. Figure 8 illustrates the latencies between data centers.

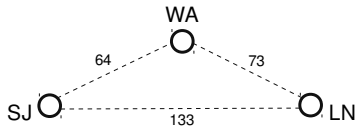


Fig. 8. Round trip time latency (ms) between the data centers of Fig. 9.

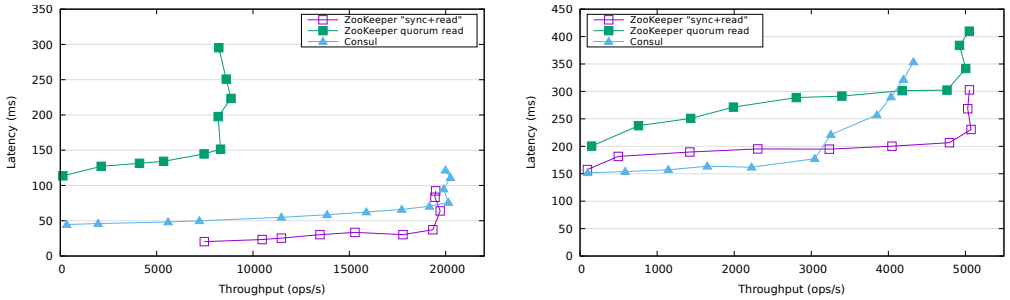
Each client machine ran up to 800 processes sequentially reading or writing 100 bytes objects to an in-memory hash map, as in Sec. 5.3. We remark that in Hybris both RMDS and clouds can be configured separately on a client basis, thus making it possible to exploit the most favorable settings in terms of deployment location. However, finding the best combination of client settings for wide area deployment requires more specific assumptions that depend on the application domain and is therefore out of the scope of this experiment. Research literature [Bezerra et al. 2014; Halalai et al. 2014] and deployment guidelines [ZooKeeper 2009; Consul 2016a] suggest mitigating the performance cost of strongly consistent wide area coordination by means of read-only servers and state partitioning. We acknowledge these approaches as beneficial and practical, despite lacking generality and a genuine cross-

[23] We deem that the most telling measure of the impact of concurrency in this setting is latency, since the throughput depends on the available bandwidth in the local RMDS cluster, which, for enterprise-grade clusters, is typically high.

data center consensus primitive. However, in this experiment we aim at assessing how far we can stretch the performance of a single consensus instance based on off-the-shelf coordination systems available as of September 2016, i.e. ZooKeeper 3.4.9 and Consul 0.7. In particular, in addition to the standard "*sync+read*" sequentially consistent ZooKeeper read, as mentioned in Sec. 4.1, we implemented a linearizable metadata read operation by prepending a quorum-based dummy write. We benchmark both the quorum-based and the *sync+read* ZooKeeper schemes along with the Consul RMDS. Figure 9 shows the results of this wide area benchmark with respect to throughput and latency performance.

During the write experiment, multiple Hybris clients performed PUT operations to different keys, while in the read experiments all clients read data associated to a single key. Both coordination systems reach peak throughput when using about 1600 concurrent clients. Note how this simple wide area deployment strategy easily reduces read throughput to $1/9$ and write throughput to $1/6$ of the corresponding figures for local clusters. Nonetheless, the throughputs and latencies recorded are arguably acceptable for a wide range of applications, especially in contexts of low write concurrency or asynchronous storage without direct user interaction.

The different performance of the two coordination systems derives from the consensus protocol they implement (i.e., Zab [Reed and Junqueira 2008] vs Raft [Ongaro and Ousterhout 2014]) and the specific API they expose. Besides, as expected, we recorded a substantial performance loss when using the quorum-based ZooKeeper reads, as they require a wider agreement among replicas. We further note that the low average latencies of the ZooKeeper *read+sync* instance are due to the presence of reads performed by clients located in the same data center of the cluster leader. Ultimately, the choice of the coordination system depends on various needs and in practice it often hinges on infrastructure already in place. Hybris accommodates these needs through its modular design that eases the implementations of different RMDS instances.



(a) GET performance (b) PUT performance

Fig. 9. Performance of metadata operations for Hybris PUT and GET in wide area settings using Consul or ZooKeeper as RMDS. Each RMDS cluster is composed by three servers deployed in San Jose, Washington D.C. and London.

5.5. Experiment 5: caching

In this experiment, we measure the performance of Hybris with and without caching (both write-through and caching-on-read simultaneously enabled). We deploy Memcached with 128MB cache limit and 10MB single object limit. We vary object sizes from 1kB to 10 MB and measure average latency using the YCSB benchmarking suite with workload B (95% reads, 5% writes). The results for GET are presented in Figure 10.

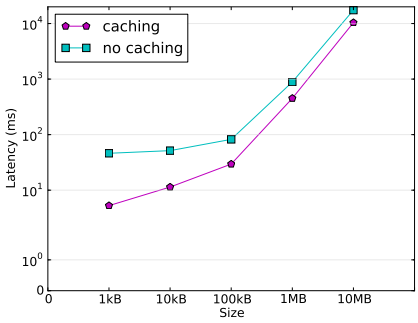


Fig. 10. Hybris GET latency with YCSB workload B (95% reads, 5% writes) varying data size.

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

5.6. Experiment 6: Hybris as personal storage backend

A thorough evaluation of a storage protocol also depends on the type of application that makes use of it. For this reason, we decided to quantify the practical benefits and overhead of using Hybris as backend of a personal storage and synchronization application. Over the last decade, this kind of application has gained a significant adoption both in household and corporate contexts. Several products have been developed and commercialized, usually as freeware in conjunction with storage pay-per-use schemes. At the same time, researchers have started studying their performance [Drago et al. 2012; Drago et al. 2013].

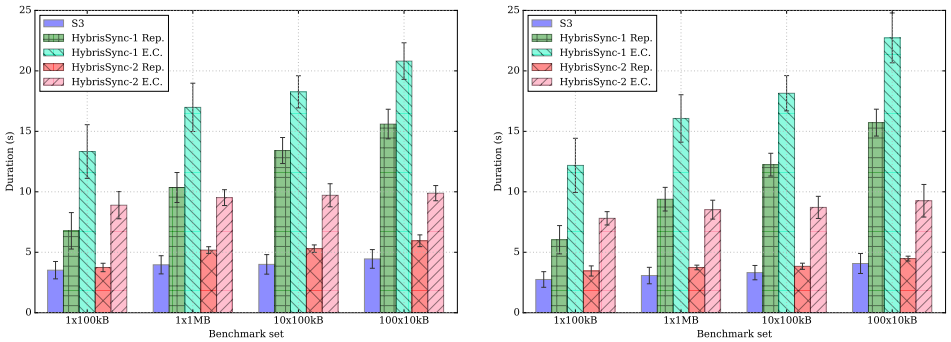
We decided to integrate Hybris as storage backend of Syncany, a popular open source storage synchronization application written in Java [Syncany 2016]. The integration entailed the development of a storage plugin in two different versions.[24] The first version, which we call *HybrisSync-1*, uses Hybris for storage of Syncany data and metadata indifferently, while the second version (*HybrisSync-2*) exposes an API to exploit Hybris RMDS also for Syncany's own metadata management. In addition, we instrumented the Syncany command line client to measure the upload and download latencies of synchronization operations. We chose to compare Hybris — in its two versions, and using replication or erasure coding — with the baseline performance of an existing storage plugin integrating Amazon S3 as remote repository.[25]

For this experiment, we hosted the RMDS on a cluster of three virtual machines as in Sec. 5.1. We employed two other similar virtual machines to simulate two clients on the same local network, mutually synchronizing the content of local folders using Syncany. During the experiment, we employed only cloud storage accounts referring to data centers located in Europe, as the client machines. Considering the statistics about workloads in personal cloud storage [Drago et al. 2012], we designed a set of benchmarks varying the number of files to be synchronized along with their sizes.

From the results shown in Fig. 11 we draw the following considerations. First, as highlighted in Sec. 5.1, erasure coding is beneficial only for object sizes that exceed a certain threshold (e.g., about 1MB in this experimental setting). For smaller ob-

[24]https://github.com/pviotti/syncany-plugin-hybris.
[25]https://github.com/syncany/syncany-plugin-s3

jects the computational overhead and the additional latency introduced by the use of a third cloud outplays the reduced payload size. Therefore, given the kind of workload involved, cross-remote storage erasure coding is not a good match for personal cloud storage. Second, the version of the Hybris plugin exposing an API for metadata management performs significantly better than the one handling in the same way both Syncany data and metadata. This is due to the high latency cost of using clouds even for lightweight operations on metadata, which, in *HybrisSync-1* are in fact stored on clouds. Finally, *HybrisSync-2* with replication perform similarly to the Amazon S3 plugin, while offering further substantial guarantees in terms of consistency and fault tolerance.



(a) Syncany upload (b) Syncany download

Fig. 11. Performance of data synchronization between hosts using different Syncany storage plugins: Amazon S3 ("S3"), and different Hybris versions. "HybrisSync-1" uses Hybris cloud storage for both Syncany data and metadata, while "HybrisSync-2" keeps Syncany's metadata in the RMDS. Both Hybris versions are evaluated in their replicated and erasure coded versions. Each bar represents the average of 30 repetitions, with whiskers marking the standard deviation interval.

In addition to this experiment, and in the scope of the CloudSpaces project [CloudSpaces 2015], we integrated Hybris with StackSync [Lopez et al. 2014], a prototype that provides, like Syncany, personal storage synchronization.[26] While exploratory, this integration demonstrates the feasibility of adopting Hybris also as backend of more scalable personal storage solutions.

5.7. Cost comparison

Table I shows an estimate of the monetary costs incurred by several cloud storage systems in the common case (i.e. in case of synchrony and without failures), including Amazon S3 as baseline. We set $f = 1$ and assume a symmetric workload that involves 10^6 write and 10^6 read operations accessing 1MB objects totaling to 1TB of storage over a period of 1 month. This corresponds to a modest workload of roughly 40 hourly operations. We further assume a metadata payload of 500B for each object. The reference costs per transaction, storage, and outbound traffic are those of the Amazon S3 US-East region,[27] as of September 5th, 2016. The cost comparison is based on the protocols' features reported in Table II, and takes into account all applicable read and

[26] https://github.com/pviotti/stacksync-desktop
[27] AWS Simple Monthly Calculator: https://calculator.s3.amazonaws.com/index.html.

write cost optimizations (e.g., preferred quorums and sequential reads). Our figures exclude the cost of the private cloud in Hybris, which we assume to be part of an already existing infrastructure.

Table I. Cost of cloud storage systems in USD for 2 x 10^6 transactions involving 10^6 objects of 1MB, totaling 1TB of storage.

System	PUT	GET	Storage Cost / Month	Total
ICStore [Basescu et al. 2012]	60	276	180	516
DepSky-A [Bessani et al. 2013]	30	93	91	214
DepSky-EC [Bessani et al. 2013]	30	93	45	168
Hybris	10	92	60	162
Hybris-EC	15	92	45	152
Amazon S3	5	92	30	127

We observe that the overhead of Hybris is twice the baseline both for PUT and storage because Hybris stores data in two clouds in the common case. Since Hybris uses 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 moderate storage overhead of 1.5 times the baseline at the cost of increased overhead for PUT, as data needs to be dispersed onto three clouds. We further note that Hybris is the most cost-effective of the multi-cloud systems considered, as it requires, in its erasure coding version, an additional expense of only 19% more than the cost of a single cloud.

6. RELATED WORK

Table II. Comparison of existing robust multi-writer cloud storage protocols. We distinguish cloud data operations (D) from cloud metadata operations (m). Unless indicated differently, properties pertain to replication-based variants.

Protocol	Semantics		Common case performance	
	Cloud faults	Consistency	No. of cloud operations	Blowup
ICStore [Basescu et al. 2012]	crash-only	linearizable[1]	$(4f+2)(D+m)$ (writes) $(2f+1)(D+m)$ (reads)	$4f+2$
DepSky [Bessani et al. 2013]	**arbitrary**	regular[1]	$(2f+1)(D+m)$ (writes) $(2f+1)(D+m)$ (reads)	$2f+1$[2]
Hybris	**arbitrary**	**linearizable**	$\mathbf{(f+1)D}$ (writes) **1D** (reads)	$\mathbf{f+1}$[3]

[1]Unlike Hybris, to achieve linearizable (resp., regular) semantics, ICStore (resp., DepSky) requires public clouds to be linearizable (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 $k > 1$.

Multi-cloud storage systems. Several storage systems have been designed to use multiple clouds to boost data robustness, notably in its reliability and availability. SafeStore [Kotla et al. 2007] erasure-codes data across multiple storage platforms (clouds) and guarantees data integrity, confidentiality and auditing. It uses a non-replicated local server as encryption proxy, and to cache data and metadata, both stored on clouds. Furthermore, SafeStore requires from cloud providers to disclose information about their internal redundancy schemes, and to expose an API that is not available in any of nowadays' cloud storage services. SPANStore [Wu et al. 2013] seeks to minimize the cost of use of multi-cloud storage, leveraging a centralized cloud placement manager. Both SafeStore and SPANStore are not robust in the Hybris sense, as their centralized components (local proxy and placement manager, respectively) are single points of failure. RACS [Abu-Libdeh et al. 2010] and HAIL [Bowers et al. 2009] assume immutable

data, hence they do not address any concurrency aspects. The Depot key-value store [Mahajan et al. 2011] tolerates any number of untrusted clouds, but does not offer strong consistency and requires computational resources on clouds.

The multi-cloud storage systems most similar to Hybris are DepSky [Bessani et al. 2013] and ICStore [Basescu et al. 2012]. For clarity, we summarize the main aspects of these systems in Table II. 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.[28] On the other hand, DepSky considers malicious clouds, yet requires $3f+1$ replicas, unlike Hybris. Furthermore, DepSky consistency guarantees are weaker than those of Hybris, even when clouds are strongly consistent. Finally, Hybris guarantees linearizability 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 [Bessani et al. 2014] 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. Nevertheless, SCFS still requires $3f + 1$ clouds to tolerate f malicious ones, i.e. the overhead it inherits from DepSky.

Latency-consistency tradeoffs for cloud storage. Numerous recent works have proposed storage systems that leverage cloud resources to implement tunable latency-consistency tradeoffs. In particular, some of these works focus on providing tunable consistency semantics expressed by static declarative contracts (e.g., [Terry et al. 2013; Sivaramakrishnan et al. 2015]) while others offer dynamic adaptive mechanisms (e.g., [Zeineddine and Bazzi 2011; Chihoub et al. 2012]). In alternative to strong consistency, Hybris provides tunable consistency semantics as well, through a static configuration of caching mechanisms implemented in trusted, private premises. Unlike previous works proposing latency-consistency tradeoffs, Hybris explicitly addresses resiliency concerns, and does not entail modification to standard cloud storage interfaces nor it requires cloud computing resources: its RMDS component has a small footprint which can be conveniently supplied by on-premises resources often already in place in corporate settings.

Separating data from metadata. The idea of separating metadata management from data storage and retrieval has been proposed in previous literature. Notably, it has been adopted in the design of parallel file systems, with the main goal of maximizing throughput (e.g., [Gibson et al. 1998; Weil et al. 2006]). Farsite [Adya et al. 2002] is an early protocol similarly proposing this design choice: it tolerates malicious faults by replicating metadata (e.g., cryptographic hashes and directory) separately from data. Hybris builds upon these techniques yet, unlike Farsite, it implements multi-writer/multi-reader semantics and is robust against timing failures as it relies on lock-free concurrency control. Furthermore, unlike Farsite, Hybris supports ephemeral clients and has no server code, targeting commodity cloud storage 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 [Ghemawat et al. 2003], 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 [Fan and Lynch 2003] and BookKeeper [Junqueira et al. 2013]. LDR [Fan and Lynch 2003] 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.

[28]The 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 file size.

However, unlike Hybris, LDR considers full-fledged servers as data storage nodes and tolerates only their crash faults. BookKeeper [Junqueira et al. 2013] implements reliable single-writer multi-reader shared storage for logs. It 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 $f + 1$ replicas and reads from a single ledger in the 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., [Fan and Lynch 2003; Junqueira et al. 2013], but also Gnothi [Wang et al. 2012]), need $2f + 1$ data repositories in the worst case, just like Hybris, which additionally tolerates arbitrary faults.

After the publication of the preliminary, conference version of this work, several follow-up storage protocols that separate metadata from data and tolerate arbitrary faults have appeared. Notably, MDStore [Cachin et al. 2014] and AWE [Androulaki et al. 2014] follow the footsteps of Hybris and use optimal number of metadata and data nodes, and implement read/write storage using replication (MDStore) and erasure coding (AWE). Unlike Hybris, MDStore and AWE are fully asynchronous and replace the eventually synchronous state-machine replication based metadata service used in Hybris with asynchronous read-write metadata service. This, however, results in increased complexity of MDStore and AWE protocols over Hybris, notably manifested in the higher latency values. Furthermore, MDStore and AWE implementations are not available, unlike that of Hybris.

More recently, Zhang et al. [2016] described the design of Cocytus, an in-memory data store that applies erasure coding to bulk data while replicating metadata and keys through a primary-backup scheme. While in Hybris we exploit data and metadata separation for fault tolerance and correctness, Cocytus adopts this hybrid scheme to enable fast data recovery. In fact, while in Cocytus data and metadata are only logically separated, Hybris store them on separate systems offering different guarantees in matter of reliability and consistency. We further note that, like Cocytus, Hybris can optionally apply erasure coding to bulk data stored on clouds.

Finally, the idea of separating control and data planes in systems tolerating arbitrary faults was used also by Yin et al. [2003] in the context of replicated state machines (RSM). While such approach 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.

Systems based on trusted components. Several systems in research literature use 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., [Correia et al. 2004; Chun et al. 2007; Kapitza et al. 2012; Veronese et al. 2013]). Some of these systems, like CheapBFT [Kapitza et al. 2012], 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 RMDS) and $f + 1$ (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 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

In this article we presented Hybris, a robust hybrid cloud storage system. Hybris scatters data (using replication or erasure coding) across multiple untrusted and possibly inconsistent public clouds, and it replicates metadata within trusted premises. Hybris

is very efficient: in the common-case, using data replication, writes involve only $f + 1$ clouds to tolerate up to f arbitrary public cloud faults, whereas reads access a single cloud. Hence, Hybris is the first multi-cloud storage protocol that makes it possible to tolerate potentially malicious clouds at the same price as coping with simple cloud outages. Furthermore, Hybris guarantees strong consistency, thanks to strongly consistent metadata stored off-clouds, used to mask the inconsistencies of cloud stores. Hybris is designed to seamlessly replace commodity key-value cloud stores (e.g., Amazon S3) in existing applications, and it can be used for storage of both archival and mutable data, due to its strong multi-writer consistency.

We also presented an extensive evaluation of the Hybris protocol. All experiments we conducted show that our system is practical, and demonstrate that it significantly outperforms comparable multi-cloud storage systems. Its performance approaches that of individual clouds.

A. ALGORITHMS AND CORRECTNESS PROOFS

This section presents pseudocode and correctness proofs for the core parts of the Hybris protocol as described in Section 3.[29] In particular, we prove that Algorithm 1, satisfies linearizability, and wait-freedom (resp. finite-write termination) for PUT (resp. for GET) operations.[30] The linearizable functionality of RMDS is specified in Alg. 2, while Alg. 3 describes the simple API required from cloud stores.

Preliminaries. 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 at line 13, Alg. 1. Else, if o is a GET, then $ts(o)$ equals the value of ts when client executes line 34, 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 $o.RMDS$ denotes calls to RMDS within operation o (and similarly for o'). Let o' be a PUT (resp. GET) operation.

Case 1 (o is a PUT): then $o.RMDS$.CONDUPDATE($o.md$) at line 23, Alg. 1, precedes (all possible calls to) $o'.RMDS$.READ() at line 29, Alg. 1 (resp., line 11, Alg. 1). By linearizability 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() at line 11, Alg. 1, where $ts \geq ts(o)$.

Case 2 (o is a GET): then since all possible calls to $o'.RMDS$.READ() at line 29 (resp. 11) follow the latest call of $o.RMDS$.READ() in line 29, by Alg. 2 and by linearizability of RMDS, it follows that $ts(o') \geq ts(o)$. If o' is a PUT, then $ts(o') > ts(o)$, similarly to Case 1. □

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

[29] For simplicity, in the pseudocode we omit the *container* parameter which can be passed as argument to Hybris APIs. Furthermore, the algorithms here presented refer to the replicated version of Hybris. The version supporting erasure codes does not entail any significant modification to algorithms and related proofs.

[30] For the sake of readability, in the proofs we ignore possible DELETE operations. However, it is easy to modify the proofs to account for their effects.

Algorithm 1 Algorithm of Hybris client cid.

Types:
  $TS = (\mathbb{N}_0 \times \mathbb{N}_0) \cup \{\perp\}$, with fields *sn* and *cid* // timestamps
  $TSMD = (TS \times (C_i \times \ldots \times C_i) \times H(V) \times \mathbb{N}_0) \cup \{\perp\}$, with fields *ts*, *replicas*, *hash* and *size*

Shared objects:
  RMDS: $MWMR$ linearizable wait-free timestamped storage object, implementing Alg. 2
  $C_{0\ldots n}$: cloud stores, exposing key-value API as in Alg. 3

Client state variables:
  $ts \in TS$, initially $(0, \perp)$
  $cloudList \in \{C_i \times \ldots \times C_i\} \cup \{\perp\}$, initially $\emptyset$

operation PUT (k, v)
  $(ts, -, -, -) \leftarrow RMDS.\text{READ}(k, false)$
  if $ts = \perp$ **then** $ts \leftarrow (0, cid)$
  $ts \leftarrow (ts.sn + 1, cid)$
  $cloudList \leftarrow \emptyset$
  trigger $timer$
  forall $f + 1$ selected clouds C_i **do**
    $C_i.put(k|ts, v)$
  wait until $|cloudList| = f + 1$ **or** $timer$ **expires**
  if $|cloudList| < f + 1$ **then**
    forall f secondary clouds C_i **do**
      $C_i.put(k|ts, v)$
    wait until $|cloudList| = f + 1$
  $RMDS.\text{CONDUPDATE}(k, ts, cloudList, H(v), size(v))$
  trigger garbage collection // see Section 3.4
  return OK

upon $put(k|ts, v)$ completes at cloud C_i
  $cloudList \leftarrow cloudList \cup \{C_i\}$

operation GET (k) // worst-case, Section 3.5 code only
  $(ts, cloudList, hash, size) \leftarrow RMDS.\text{READ}(k, true)$
  if $ts = \perp$ **or** $cloudList = \perp$ **then return** $\perp$
  forall $C_i \in cloudList$ **do**
    $C_i.get(k|ts)$

upon $get(k|ts)$ returns $data$ from cloud C_i
  if $H(data) = hash$ **then return** $data$
  else $C_i.get(k|ts)$

upon received $notify(k, ts')$ from RMDS such that $ts' > ts$
  cancel all pending *get*
  return GET (k)

operation LIST ()
  $mdList \leftarrow RMDS.\text{LIST}()$
  forall $md \in mdList$ **do**
    if $md.cloudList = \perp$ **then**
      $mdList \leftarrow mdList \setminus \{md\}$
  return $mdList$

operation DELETE (k)
  $(ts, cloudList, -, -) \leftarrow RMDS.\text{READ}(k, false)$
  if $ts = \perp$ **or** $cloudList = \perp$ **then return** OK
  $ts \leftarrow (ts.sn + 1, cid)$
  $RMDS.\text{CONDUPDATE}(k, ts, \perp, \perp, 0)$
  trigger garbage collection // see Section 3.4
  return OK

Algorithm 2 RMDS functionality (linearizable).

52: **Server state variables:**
53: $md \subseteq K \times TSMD$, initially $\perp$, read and written through $mdf : K \rightarrow TSMD$
54: $sub \subseteq K \times (\mathbb{N}_0 \times \ldots \times \mathbb{N}_0)$, initially $\perp$, read and written through $subf : K \rightarrow (\mathbb{N}_0 \times \ldots \times \mathbb{N}_0)$

55: **operation** CONDUPDATE $(k, ts, cList, hash, size)$
56: $(ts_k, -, -, -) \leftarrow mdf(k)$
57: **if** $ts_k = \perp$ **or** $ts > ts_k$ **then**
58: $mdf : k \leftarrow (ts, cList, hash, size)$
59: send **notify**(k, ts) to every $cid \in subf(k)$
60: $subf : k \leftarrow \emptyset$
61: **return** OK

62: **operation** READ $(k, subscribe)$ by cid
63: **if** $subscribe$ **then**
64: $subf : k \leftarrow subf(k) \cup \{cid\}$
65: **return** $mdf(k)$

66: **operation** LIST ()
67: **return** $mdf(*)$

Algorithm 3 Cloud store C_i functionality.

68: **Server state variables:**
69: $data \subseteq K \times V$, initially $\emptyset$, read and written through $f : K \rightarrow V$

70: **operation** *put*(key, val)
71: $f : key \leftarrow value$
72: **return** OK

73: **operation** *get*(key)
74: **return** $f(key)$

PROOF. By lines 11-13, 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 $v \neq \perp$. Then there exists 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 at line 34 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 17. 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 linearizability.*

PROOF. Let ex be an execution of Algorithm 1. By Lemma A.3 the timestamp of a GET either has been written by some PUT or the GET returns $\perp$. 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 $\perp$, according to their timestamps and by placing all GET operations that did

not return $\perp$ immediately after the PUT operation with the same timestamp. The GET operations that did return $\perp$ are placed in the beginning of π.

Towards linearizability, 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 $\perp$ if there is no such PUT. In the latter case, by construction rd is ordered before any PUT in π. Otherwise, $v \neq \perp$ 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 operations follows from: a) the assumption of using $2f + 1$ clouds out of which at most f may be faulty (and hence the *wait* statement at line 22, Alg. 1 is non-blocking), and b) wait-freedom of calls to RMDS (hence, calls to RMDS at lines 11 and 23, 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 at line 23, Alg. 1. We distinguish two cases: (i) rd invokes an infinite number of recursive GET calls (in line 38, Alg 1), and (ii) rd never passes the check at line 34, 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 38, 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 $cloudList$ at line 23, Alg. 1 in wr. Since rd does not terminate, it receives a notification at line 36, 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 at line 34, Alg. 1 and rd completes. A contradiction. □

ACKNOWLEDGMENTS

We thank Christian Cachin and Phillip B. Gibbons for their comments on earlier drafts of this article.

REFERENCES

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

Hussam Abu-Libdeh, Lonnie Princehouse, and Hakim Weatherspoon. 2010. RACS: a case for cloud storage diversity. In *ACM Symposium on Cloud Computing (SoCC)*. 229–240. http://doi.acm.org/10.1145/1807128.1807165

Atul Adya, William J. Bolosky, Miguel Castro, Gerald Cermak, Ronnie Chaiken, John R. Douceur, Jon Howell, Jacob R. Lorch, Marvin Theimer, and Roger Wattenhofer. 2002. FARSITE: Federated, Available, and

Reliable Storage for an Incompletely Trusted Environment. In *Symposium on Operating System Design and Implementation OSDI*. http://www.usenix.org/events/osdi02/tech/adya.html

Amazon. 2016. Amazon DynamoDB Pricing. (2016). Available online at https://aws.amazon.com/dynamodb/pricing/; visited September 2016.

Elli Androulaki, Christian Cachin, Dan Dobre, and Marko Vukolic. 2014. Erasure-Coded Byzantine Storage with Separate Metadata. In *Principles of Distributed Systems - OPODIS 2014*. 76–90. DOI:http://dx.doi.org/10.1007/978-3-319-14472-6_6

Apache. 2016. Apache JClouds. (2016). Available online at http://jclouds.apache.org/; visited September 2016.

Michael Armbrust, Armando Fox, Rean Griffith, Anthony D. Joseph, Randy H. Katz, Andy Konwinski, Gunho Lee, David A. Patterson, Ariel Rabkin, Ion Stoica, and Matei Zaharia. 2010. A view of cloud computing. *Commun. ACM* 53, 4 (2010), 50–58. DOI:http://dx.doi.org/10.1145/1721654.1721672

Peter Bailis and Ali Ghodsi. 2013. Eventual consistency today: limitations, extensions, and beyond. *Commun. ACM* 56, 5 (2013), 55–63. http://doi.acm.org/10.1145/2447976.2447992

Cristina Basescu, Christian Cachin, Ittay Eyal, Robert Haas, Alessandro Sorniotti, Marko Vukolić, and Ido Zachevsky. 2012. Robust data sharing with key-value stores. In *IEEE/IFIP International Conference on Dependable Systems and Networks, DSN*. 1–12. DOI:http://dx.doi.org/10.1109/DSN.2012.6263920

David Bermbach and Stefan Tai. 2014. Benchmarking Eventual Consistency: Lessons Learned from Long-Term Experimental Studies. In *IEEE International Conference on Cloud Engineering (IC2E)*. 47–56. DOI:http://dx.doi.org/10.1109/IC2E.2014.37

Alysson Bessani, Ricardo Mendes, Tiago Oliveira, Nuno Neves, Miguel Correia, Marcelo Pasin, and Paulo Verissimo. 2014. SCFS: A Shared Cloud-backed File System. In *USENIX Annual Technical Conference, USENIX ATC*.

Alysson Neves Bessani, Miguel Correia, Bruno Quaresma, Fernando André, and Paulo Sousa. 2013. DepSky: Dependable and Secure Storage in a Cloud-of-Clouds. *ACM Transactions on Storage* 9, 4 (2013), 12. http://doi.acm.org/10.1145/2535929

Carlos Eduardo Benevides Bezerra, Fernando Pedone, and Robbert van Renesse. 2014. Scalable State-Machine Replication. In *IEEE/IFIP International Conference on Dependable Systems and Networks, DSN*. 331–342. DOI:http://dx.doi.org/10.1109/DSN.2014.41

Kevin D. Bowers, Ari Juels, and Alina Oprea. 2009. HAIL: a high-availability and integrity layer for cloud storage. In *ACM CCS*. 187–198. http://doi.acm.org/10.1145/1653662.1653686

Eric A. Brewer. 2012. Pushing the CAP: Strategies for Consistency and Availability. *IEEE Computer* 45, 2 (2012), 23–29. http://doi.ieeecomputersociety.org/10.1109/MC.2012.37

Christian Cachin, Dan Dobre, and Marko Vukolic. 2014. Separating Data and Control: Asynchronous BFT Storage with 2t + 1 Data Replicas. In *Stabilization, Safety, and Security of Distributed Systems SSS*. 1–17. DOI:http://dx.doi.org/10.1007/978-3-319-11764-5_1

Ignacio Cano, Srinivas Aiyar, and Arvind Krishnamurthy. 2016. Characterizing Private Clouds: A Large-Scale Empirical Analysis of Enterprise Clusters. In *ACM Symposium on Cloud Computing (SoCC)*. 29–41. DOI:http://dx.doi.org/10.1145/2987550.2987584

Houssem-Eddine Chihoub, Shadi Ibrahim, Gabriel Antoniu, and María S. Pérez-Hernández. 2012. Harmony: Towards Automated Self-Adaptive Consistency in Cloud Storage. In *IEEE International Conference on Cluster Computing, CLUSTER*. 293–301. DOI:http://dx.doi.org/10.1109/CLUSTER.2012.56

Gregory Chockler, Dan Dobre, and Alexander Shraer. 2013. Brief Announcement: Consistency and Complexity Tradeoffs for Highly-Available Multi-Cloud Store. In *The International Symposium on Distributed Computing (DISC)*.

Byung-Gon Chun, Petros Maniatis, Scott Shenker, and John Kubiatowicz. 2007. Attested append-only memory: making adversaries stick to their word. In *SOSP*. 189–204. http://doi.acm.org/10.1145/1294261.1294280

CloudSpaces. 2015. CloudSpaces EU FP7 project. (2015). Available online at http://cloudspaces.eu/; visited September 2016.

Consul. 2016a. Consul - consensus protocol. (2016). Available online at https://www.consul.io/docs/internals/consensus.html; visited September 2016.

Consul. 2016b. Consul - distributed service discovery and configuration. (2016). Available online at https://www.consul.io/; visited September 2016.

Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. 2010. Benchmarking cloud serving systems with YCSB. In *ACM Symposium on Cloud Computing (SoCC)*. 143–154. http://doi.acm.org/10.1145/1807128.1807152

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. 2013. Spanner: Google's Globally Distributed Database. *ACM Trans. Comput. Syst.* 31, 3 (2013), 8. http://doi.acm.org/10.1145/2491245

Miguel Correia, Daniel Gómez Ferro, Flavio Paiva Junqueira, and Marco Serafini. 2012. Practical Hardening of Crash-Tolerant Systems. In *USENIX Annual Technical Conference*. https://www.usenix.org/conference/atc12/technical-sessions/presentation/correia

Miguel Correia, Nuno Ferreira Neves, and Paulo Veríssimo. 2004. How to Tolerate Half Less One Byzantine Nodes in Practical Distributed Systems. In *SRDS*. 174–183. http://doi.ieeecomputersociety.org/10.1109/RELDIS.2004.1353018

Idilio Drago, Enrico Bocchi, Marco Mellia, Herman Slatman, and Aiko Pras. 2013. Benchmarking personal cloud storage. In *ACM SIGCOMM Internet Measurement Conference, IMC 2013*. 205–212. DOI:http://dx.doi.org/10.1145/2504730.2504762

Idilio Drago, Marco Mellia, Maurizio M. Munafò, Anna Sperotto, Ramin Sadre, and Aiko Pras. 2012. Inside dropbox: understanding personal cloud storage services. In *ACM SIGCOMM Internet Measurement Conference, IMC '12*. 481–494. DOI:http://dx.doi.org/10.1145/2398776.2398827

Cynthia Dwork, Nancy Lynch, and Larry Stockmeyer. 1988. Consensus in the presence of partial synchrony. *J. ACM* 35, 2 (April 1988), 288–323. DOI:http://dx.doi.org/10.1145/42282.42283

Eurecom. 2016. Hybris - robust hybrid cloud storage library. (2016). Available online at https://github.com/pviotti/hybris; visited September 2016.

Rui Fan and Nancy Lynch. 2003. Efficient Replication of Large Data Objects. In *DISC*. 75–91.

Aishwarya Ganesan, Ramnatthan Alagappan, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. 2017. Redundancy Does Not Imply Fault Tolerance: Analysis of Distributed Storage Reactions to Single Errors and Corruptions. In *USENIX Conference on File and Storage Technologies (FAST 17)*. https://www.usenix.org/conference/fast17/technical-sessions/presentation/ganesan

Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. 2003. The Google file system. In *SOSP*. 29–43. http://doi.acm.org/10.1145/945445.945450

Garth A. Gibson, David Nagle, Khalil Amiri, Jeff Butler, Fay W. Chang, Howard Gobioff, Charles Hardin, Erik Riedel, David Rochberg, and Jim Zelenka. 1998. A Cost-Effective, High-Bandwidth Storage Architecture. In *International Conference on Architectural Support for Programming Languages and Operating Systems*. 92–103. DOI:http://dx.doi.org/10.1145/291069.291029

Seth Gilbert and Nancy A. Lynch. 2002. Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. *SIGACT News* 33, 2 (2002), 51–59. http://doi.acm.org/10.1145/564585.564601

Wojciech M. Golab, Xiaozhou Li, and Mehul A. Shah. 2011. Analyzing consistency properties for fun and profit. In *ACM Symposium on Principles of Distributed Computing (PODC), 2011*. 197–206.

Haryadi S. Gunawi, Mingzhe Hao, Riza O. Suminto, Agung Laksono, Anang D. Satria, Jeffry Adityatama, and Kurnia J. Eliazar. 2016. Why Does the Cloud Stop Computing?: Lessons from Hundreds of Service Outages. In *ACM Symposium on Cloud Computing (SoCC)*. 1–16. DOI:http://dx.doi.org/10.1145/2987550.2987583

Raluca Halalai, Pierre Sutra, Etienne Riviere, and Pascal Felber. 2014. ZooFence: Principled Service Partitioning and Application to the ZooKeeper Coordination Service. In *IEEE International Symposium on Reliable Distributed Systems*. 67–78. DOI:http://dx.doi.org/10.1109/SRDS.2014.41

James Hamilton. 2009. The cost of latency. (2009). Available online at http://perspectives.mvdirona.com/2009/10/the-cost-of-latency/; visited September 2016.

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

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

Patrick Hunt, Mahadev Konar, Flavio P. Junqueira, and Benjamin Reed. 2010. ZooKeeper: wait-free coordination for internet-scale systems. In *USENIX Annual Technical Conference, USENIX ATC*. 11–11.

Flavio Paiva Junqueira, Ivan Kelly, and Benjamin Reed. 2013. Durability with BookKeeper. *Operating Systems Review* 47, 1 (2013), 9–15. http://doi.acm.org/10.1145/2433140.2433144

Rüdiger Kapitza, Johannes Behl, Christian Cachin, Tobias Distler, Simon Kuhnle, Seyed Vahid Mohammadi, Wolfgang Schröder-Preikschat, and Klaus Stengel. 2012. CheapBFT: resource-efficient Byzantine fault tolerance. In *EuroSys*. 295–308. http://doi.acm.org/10.1145/2168836.2168866

Ramakrishna Kotla, Lorenzo Alvisi, and Michael Dahlin. 2007. SafeStore: A Durable and Practical Storage System. In *USENIX Annual Technical Conference, USENIX ATC*. 129–142.

Leslie Lamport. 1979. How to Make a Multiprocessor Computer That Correctly Executes Multiprocess Programs. *IEEE Trans. Comput.* 28, 9 (1979), 690–691.

Shengyun Liu, Paolo Viotti, Christian Cachin, Vivien Quéma, and Marko Vukolic. 2016. XFT: Practical Fault Tolerance beyond Crashes. In *USENIX Symposium on Operating Systems Design and Implementation, OSDI*. https://www.usenix.org/conference/osdi16/technical-sessions/presentation/liu

Pedro Garcia Lopez, Sergi Toda, Cristian Cotes, Marc Sanchez-Artigas, and John Lenton. 2014. StackSync: Bringing Elasticity to Dropbox-like File Synchronization. In *ACM/IFIP/USENIX Middleware*.

Nancy A. Lynch and Alexander A. Shvartsman. 2002. RAMBO: A Reconfigurable Atomic Memory Service for Dynamic Networks. In *DISC*. 173–190.

Prince Mahajan, Srinath T. V. Setty, Sangmin Lee, Allen Clement, Lorenzo Alvisi, Michael Dahlin, and Michael Walfish. 2011. Depot: Cloud Storage with Minimal Trust. *ACM Trans. Comput. Syst.* 29, 4 (2011), 12.

Memcached. 2016. Memcached. (2016). Available online at http://memcached.org/; visited September 2016.

Diego Ongaro and John K. Ousterhout. 2014. In Search of an Understandable Consensus Algorithm. In *USENIX Annual Technical Conference, USENIX ATC*. 305–319. https://www.usenix.org/conference/atc14/technical-sessions/presentation/ongaro

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

J. S. Plank, S. Simmerman, and C. D. Schuman. 2008. *Jerasure: A Library in C/C++ Facilitating Erasure Coding for Storage Applications - Version 1.2*. Technical Report CS-08-627. University of Tennessee.

Benjamin Reed and Flavio P. Junqueira. 2008. A Simple Totally Ordered Broadcast Protocol. In *Workshop on Large-Scale Distributed Systems and Middleware (LADIS)*. 2:1–2:6. DOI:http://dx.doi.org/10.1145/1529974.1529978

Rodrigo Rodrigues and Barbara Liskov. 2005. High Availability in DHTs: Erasure Coding vs. Replication. In *IPTPS*. 226–239. http://dx.doi.org/10.1007/11558989_21

K. C. Sivaramakrishnan, Gowtham Kaki, and Suresh Jagannathan. 2015. Declarative programming over eventually consistent data stores. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*. 413–424. DOI:http://dx.doi.org/10.1145/2737924.2737981

SoftLayer. 2016. IBM SoftLayer. (2016). Available online at http://www.softlayer.com/; visited September 2016.

Emil Stefanov, Marten van Dijk, Elaine Shi, Christopher W. Fletcher, Ling Ren, Xiangyao Yu, and Srinivas Devadas. 2013. Path ORAM: an extremely simple oblivious RAM protocol. In *ACM SIGSAC Conference on Computer and Communications Security, (CCS'13)*. http://doi.acm.org/10.1145/2508859.2516660

Syncany. 2016. Secure file synchronization software for arbitrary storage backends. (2016). Available online at https://www.syncany.org/; visited September 2016.

Douglas B. Terry, Alan J. Demers, Karin Petersen, Mike Spreitzer, Marvin Theimer, and Brent B. Welch. 1994. Session Guarantees for Weakly Consistent Replicated Data. In *Parallel and Distributed Information Systems (PDIS)*. 140–149.

Douglas B. Terry, Vijayan Prabhakaran, Ramakrishna Kotla, Mahesh Balakrishnan, Marcos K. Aguilera, and Hussam Abu-Libdeh. 2013. Consistency-based service level agreements for cloud storage. In *ACM SIGOPS Symposium on Operating Systems Principles, SOSP*. 309–324. DOI:http://dx.doi.org/10.1145/2517349.2522731

Francisco J. Torres-Rojas, Mustaque Ahamad, and Michel Raynal. 1999. Timed Consistency for Shared Distributed Objects. In *ACM Symposium on Principles of Distributed Computing (PODC), 1999*. 163–172.

Giuliana Santos Veronese, Miguel Correia, Alysson Neves Bessani, Lau Cheuk Lung, and Paulo Veríssimo. 2013. Efficient Byzantine Fault-Tolerance. *IEEE Trans. Computers* 62, 1 (2013), 16–30.

Paolo Viotti and Marko Vukolic. 2016. Consistency in Non-Transactional Distributed Storage Systems. *ACM Comput. Surv.* 49, 1 (2016), 19. DOI:http://dx.doi.org/10.1145/2926965

VMware. 2013. The Snowden Leak: A Windfall for Hybrid Cloud? (2013). Available online at http://blogs.vmware.com/consulting/2013/09/the-snowden-leak-a-windfall-for-hybrid-cloud.html; visited September 2016.

Werner Vogels. 2009. Eventually consistent. *Commun. ACM* 52, 1 (2009), 40–44. http://doi.acm.org/10.1145/1435417.1435432

Marko Vukolić. 2010. The Byzantine empire in the intercloud. *SIGACT News* 41, 3 (2010), 105–111. DOI:http://dx.doi.org/10.1145/1855118.1855137

Yang Wang, Lorenzo Alvisi, and Mike Dahlin. 2012. Gnothi: separating data and metadata for efficient and available storage replication. In *USENIX Annual Technical Conference, USENIX ATC*. 38–38.

Hakim Weatherspoon and John Kubiatowicz. 2002. Erasure Coding Vs. Replication: A Quantitative Comparison. In *IPTPS*. 328–338.

Sage A. Weil, Scott A. Brandt, Ethan L. Miller, Darrell D. E. Long, and Carlos Maltzahn. 2006. Ceph: A Scalable, High-Performance Distributed File System. In *Symposium on Operating Systems Design and Implementation (OSDI)*. 307–320. http://www.usenix.org/events/osdi06/tech/weil.html

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

Jian Yin, Jean-Philippe Martin, Arun Venkataramani, Lorenzo Alvisi, and Michael Dahlin. 2003. Separating agreement from execution for byzantine fault tolerant services. In *SOSP*. 253–267. http://doi.acm.org/10.1145/945445.945470

Hassan Zeineddine and Wael Bazzi. 2011. Rationing data updates with consistency considerations in distributed systems. In *IEEE International Conference on Networks, ICON*. 165–170. DOI:http://dx.doi.org/10.1109/ICON.2011.6168469

Heng Zhang, Mingkai Dong, and Haibo Chen. 2016. Efficient and Available In-memory KV-Store with Hybrid Erasure Coding and Replication. In *USENIX Conference on File and Storage Technologies, FAST*. https://www.usenix.org/conference/fast16/technical-sessions/presentation/zhang-heng

Observers ZooKeeper. 2009. Observers - making ZooKeeper scale even further. (2009). Available online at https://blog.cloudera.com/blog/2009/12/observers-making-zookeeper-scale-even-further/; visited September 2016.

Received Month Year; revised Month Year; accepted Month Year