SEVENTH FRAMEWORK PROGRAMME

CloudSpaces

(FP7-ICT-2011-8)

Open Service Platform for the Next Generation of Personal Clouds

D3.1 Guidelines for Heterogeneous Personal Clouds

Due date of deliverable: 30-11-2013
Actual submission date: 12-11-2013

Start date of project: 01-10-2012 Duration: 36 months

Summary of the document

Document Type	Deliverable
Dissemination level	Public
State	Final
Number of pages	28
WP/Task related to this document	WP3
WP/Task responsible	EUR
Author(s)	Refer to contributors list
Partner(s) Contributing	EUR, URV
Document ID	CLOUDSPACES_D3.1_131112_Public.pdf
Abstract	Guidelines on adaptive distributed synchronization and replication schemes for heterogeneous Personal Clouds. Early prototype of the Adaptive edge platform (adaptors, mediator). Deployment and management of the Adaptive cloud platform testbed (Openstack Swift, massive testbed, first user trials and distributed testing) Architecture, APIs, and use cases of Adaptive Proximity-aware services.
Keywords	adaptive synchronization, adaptive replication, Personal Clouds, adaptors, mediators, proximity-aware

Contributors

Name	Last name	Affiliation	Email
Marko	Vukolić	EUR	marko.vukolic@eurecom.fr
Pedro	García López	URV	pedro.garcia@urv.cat
Xavi	León	URV	xavi.leon@urv.cat

Table of Contents

1 Executive summary

Personal clouds are inherently heterogeneous. Companies and home users store their data on a variety of private devices as well as on different clouds. In Cloudspaces WP3 we approach the problem of heterogeneity in personal clouds from two angles: (i) heterogeneous public clouds and (ii) heterogeneous clients.

In the context of heterogeneous public clouds we propose a robust hybrid cloud storage solution (HCS). Our HCS system provides a robust and efficient storage abstraction over multiple clouds that can be used as Personal Cloud backend in Cloudspaces. On the other hand, in the context of heterogeneous clients, we propose an Adaptive edge platform based on BitTorrent.

Early prototypes of both approaches are presented and explained in details. Our early prototypes are founded on and implicitly contain a number of design guidelines for dealing with cloud heterogeneity.

2 Introduction

Personal clouds are inherently heterogeneous. Companies and home users store their data on a variety of private devices as well as on different clouds.

In Cloudspaces WP3 we approach the problem of heterogeneity in personal clouds from two angles:

- Hybrid cloud storage (HCS). We designed multi-cloud storage backend that orchestrates heterogeneous public clouds. Our HCS system provides a robust and efficient storage abstraction over multiple clouds that can be used as Personal Cloud backend in Cloudspaces. As such, the home users and SMEs can gain more control over their data. Early prototype design of HCS, designed by EUR, is described in Section 3.
- Adaptive edge platform. The variety of different, heterogeneous users of personal clouds can be leveraged to overcome current high cost of data transfers in current cloud providers. To this end, URV has designed an adaptive edge platform that replaces regular HTTP cloud unicasts with BitTorrent [1]. BitTorrent effectively leverages swarms of heterogenous personal cloud users to boost the efficiency of personal clouds. Early prototype design of Cloudspaces Adaptive edge platform is described in Section 4. This adaptive platform prototype has been massively deployed by two Cloudspaces partners: URV and TISSAT. We discuss the deployment of adaptive platforms in details in Section 4.4.

Our two approaches to heterogeneity can be seen as complementary: HCS explores heterogeneity of cloud storage resources, whereas adaptive edge platform leverages heterogeneous clients. Our early prototypes are founded on and implicitly contain a number of design guidelines for dealing with heterogeneity in personal clouds. In this deliverable, instead of giving an itemized list of independent design guidelines, we opt to present our guidelines for dealing with cloud heterogeneity by integrating those guidelines in proposals for early system-level solutions (i.e,. HCS and adaptive edge platform) that we will develop in Cloudspaces WP3.

3 Early prototype of Hybrid Cloud Storage

3.1 Motivation for Hybrid Cloud Storage

Most of the personal cloud storage providers such as Dropbox, SygarSync, Box or others, ultimately store users' data on proprietary commodity cloud storage services such as Amazon S3. In this way, users are given no control whatsoever over where their data is stored and have no efficient means to migrate their data to a different commodity cloud service other than change the personal cloud storage provider altogether. Even then, chances are data are going to end up at the same commodity cloud storage provider anyway, causing data gravity and provider lock-in.

Clearly, data gravity and provider lock-in have serious implications for privacy, availability, performance and cost of using personal cloud services. To this end in WP3 of CloudSpaces (Cloudspaces Storage) we will focus on ways to provide adaptive replication and synchronization for personal storage infrastructure that will integrate different, heterogeneous user and Cloud storage resources and avoid data lock-in. Our focal use case is specifically that of an SME that wishes to use (personal) cloud services, but wants more control over its own data.

Such an SME, desiring to use public cloud service, typically owns some computation and storage resources in a private infrastructure. A high-level solution for such a company is hybrid cloud storage which entails storing data on private premises as well as on one (or more) remote, public cloud storage providers. As such, hybrid cloud storage brings to enterprises the benefits of public cloud storage (e.g., elasticity, flexible payment schemes and disaster-safe durability) while allowing the enterprises to maintain the control over their data. For example, an enterprise can keep the sensitive data on premises while storing less sensitive data at potentially untrusted public clouds. In a sense, hybrid cloud eliminates to a large extent the concerns that companies have with trusting their data to commercial clouds [2] — as a result, enterprise-class hybrid cloud storage solutions are booming with all leading storage providers such as EMC [3], IBM [4], NetApp [5], Microsoft [6] and others offering their proprietary solutions. Such solutions are often very costly and are arguably beyond reach of a typical SME.

As an alternative approach to solving trust and reliability concerns associated with public cloud storage providers, several research works (e.g., [7, 8, 9]) considered storing data robustly into public clouds, by leveraging multiple cloud providers (i.e,. without relying on private resources). In short, the ideas behind these public multi-cloud storage systems such as Depsky [7], ICStore [8, 10] and CSpan [9] is to distribute the trust across several public clouds, and/or increase data reliability and address vendor lock-in concerns. A significant advantage of this approach, that makes it also interesting for SMEs, is that it is typically based on client libraries that share data accessing commodity clouds, and as such, demands no big investments into proprietary storage solutions. However, none of these robust multi-cloud storage systems considered leveraging resources on private premises, with non-negligible compute and storage capacity that is arguably present today in many SMEs and even in many households.

Our main guideline to designing novel cloud storage solutions with heterogeneous resources is to leverage private portion of these resources to store personal cloud metadata, to

give a user control over data stored in a public cloud.

In this deliverable we present the key aspects of HCS, the first robust hybrid cloud storage system. The key idea behind HCS is that it keeps all storage metadata on private premises, even when those metadata pertain to data outsourced to public clouds. This separation of metadata from data allows HCS to significantly outperform existing robust public multi-cloud storage systems, both in terms of system performance (e.g., latency) and storage cost while providing strong consistency guarantees (namely, linearizability [11]). The key features of HCS can be summarized as follows:

- HCS is a multi-writer multi-reader key-value storage system that guarantees linearizability (atomicity) of reads and writes even in presence of eventually consistent public clouds. To this end, HCS introduces consistency hardening, i.e., HCS leverages the atomicity of metadata stored locally on premises to mask the possible inconsistencies of public clouds [12, 13].
- HCS puts no trust in any given public cloud provider; namely, HCS can mask arbitrary (including malicious) faults of up to f public clouds. However, unlike traditional storage systems that involve $3f + 1$ storage nodes to mask f malicious ones (e.g., [7]), HCS is the first storage system that involves only $2f + 1$ public clouds in the worst case to mask f malicious ones. This feature is a byproduct of HCS design which separates metadata from data which, as HCS demonstrates, ripes capital benefits in tolerating untrusted data repositories.
- HCS is efficient and incurs low cost. In common case, when the system is synchronous and without faults, HCS write to public clouds involves as few as $f + 1$ public clouds, whereas reads involve only a single cloud, despite the fact that clouds are untrusted. HCS achieves this without relying on expensive cryptographic primitives; indeed, in masking malicious faults, HCS relies solely on cryptographic hashes.
- HCS metadata is stored on private premises in a fault-tolerant manner, where faults on private premises are assumed to be crash-only. To maintain the HCS footprint small and to facilitate its adoption, we chose to replicate HCS metadata layering HCS on top of Apache Zookeeper [14, 15]. HCS clients act simply as Zookeeper client — our system does not entail any modifications to Zookeeper, hence facilitating HCS deployment.
- HCS supports file-based deduplication and caching of data stored at public clouds (without impacting consistency). While different caching solutions can be applied to HCS, we chose to interface HCS with established distributed cache such as memcached [16], where memcached is deployed on the same machines that run Zookeeper servers. Beyond caching data stored on public clouds, HCS allows seamless integration with different key-value stores (e.g., Cassandra, HBase, Redis) that a company may use on private premises for storing more sensitive data.

Compared to existing cloud storage services, HCS provides the better reliability, performance (e.g., latency), consistency, as well as inherent tolerance of untrusted cloud repositories. HCS adapts to heterogeneous networks and topologies, leveraging geographical distribution of public commodity storage clouds. In future, HCS will also allow dynamic cloud reconfiguration inherently fighting data gravity and provider lock-in and support erasure-coding.

3.2 HCS Architecture

3.2.1 Overview

High-level design of HCS is given in Figure 1. HCS mixes two types of resources: 1) private, trusted resources that consist of computation and (limited) storage resources and 2) public (and virtually unlimited) untrusted storage resources in the clouds. HCS is designed to leverage public cloud storage repositories whose API does not offer computation. A de-facto standard for this type of storage interface is a key-value store (e.g., Amazon S3 [17] or Openstack Swift [18]).

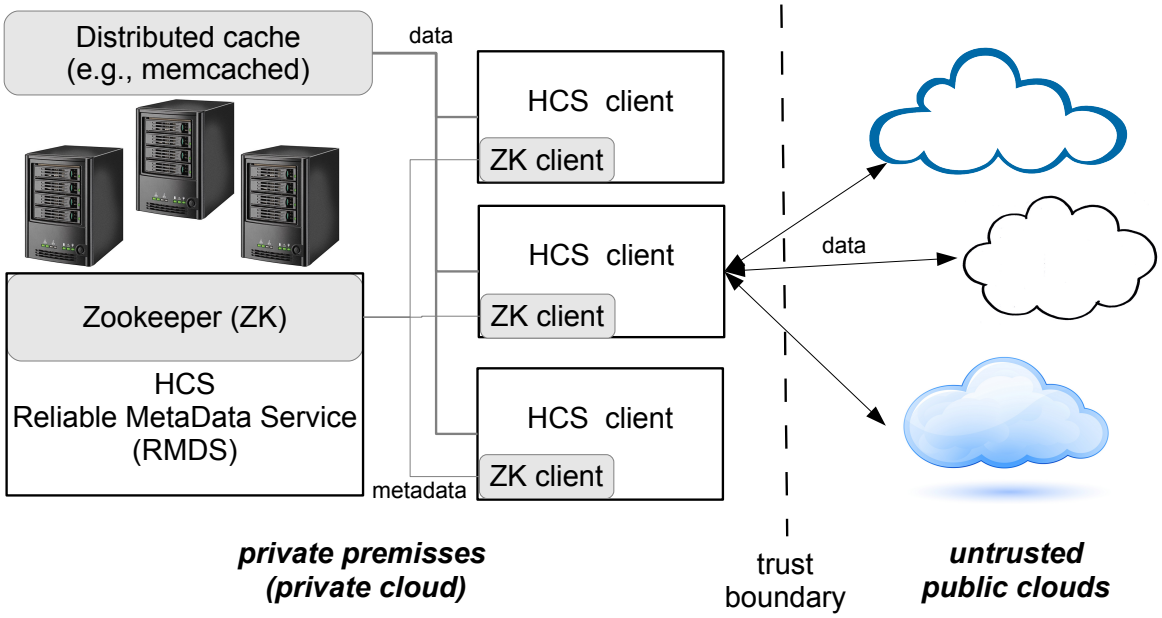


Figure 1: HCS architecture. Reused (open-source) components are depicted in grey.

HCS stores metadata separately from public cloud data. Metadata is stored within the key component of HCS called Reliable MetaData Service (RMDS). RMDS has no single point of failure and, in our implementation, resides on private premises. Our design (see Fig. 1) does not restrict RMDS to a specific implementation; indeed, RMDS can be implemented using replicated databases (e.g., MySQL), NoSQL replicated data stores with conditional updates (e.g., HBase, MongoDB), or using a custom replicated state machine (e.g., Paxos [19]). However, our goal is an easily deployable and maintainable, yet scalable and reliable storage system, that can be easily adopted. Hence, we chose to implement RMDS as a thin layer on top of the Apache Zookeeper coordination service [14, 15] (see Sec. 3.4 for details). Our choice is driven by the fact that Zookeeper is invented with similar applications in mind; moreover, Zookeeper is already a part of the open source Hadoop stack, hence likely to be familiar to companies interested in deploying private cloud infrastructure. Zookeeper-based RMDS supports HCS scalability: HCS can scale to thousands of clients.

On the other hand, HCS stores data (mainly) on untrusted public clouds. Data is replicated across multiple cloud storage providers for robustness, i.e., to mask cloud outages and

even malicious faults. In addition to storing data and public clouds, HCS architecture supports data caching on private premises. While different caching solutions can be used, our HCS implementation reuses memcached [16], an open source distributed caching system. In our implementation/deployment, we collocate memcached and Zookeeper servers (see Sec. 3.4 for details), but this is merely one possible choice.

Finally, a HCS client is at the heart of the system. HCS client library is responsible for interactions with public clouds, RMDS and the caching service. Hence, in our implementation, HCS client also contains the Zookeeper client for interactions with RDMS. HCS clients are also responsible for encrypting and decrypting data in case data confidentiality is enabled — in this case, clients leverage RMDS for sharing encryption keys (see Sec. 3.3.7).

In the rest of this section, we first define our system model and assumptions in more details. Then we define HCS data model and specify its consistency and liveness semantics.

3.2.2 System Model

We assume an unreliable distributed system where any of the components might fail. In particular, we consider a dual or hybrid [20] fault model, where: (i) the processes on private premises (i.e., in the private cloud) can fail by crashing, and (ii) we model public clouds as potentially malicious (i.e., arbitrary-fault prone [21]) processes. Processes that do not fail are called correct.

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

Similarly to our fault model, our communication model is dual, with the model boundary coinciding with our trust boundary (see Fig. 1).[1] Namely, we assume that the communication among processes located in the private portion of the cloud is partially synchronous [22] (i.e., with arbitrary but finite periods of asynchrony), whereas the communication among clients and public clouds is entirely asynchronous.

Our consistency model is likewise dual. We model processes on private premises as strongly consistent, with their computation proceeding in indivisible, atomic steps. On the other hand, we model clouds as eventually consistent [12]; informally, eventual consistency guarantees that, if no new updates are made to a given data item, eventually all accesses to that item will return the last updated value.

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

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

3.2.3 Data model and semantics

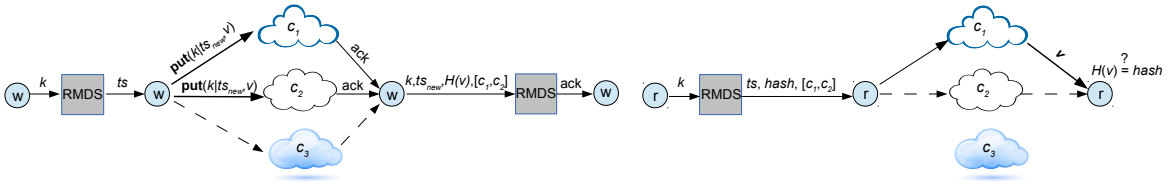
Similarly to commodity public cloud storage services, HCS exports a key-value store (KVS) API; in particular, HCS address space consists of flat containers, each holding multiple keys. The KVS API features four main operations: (i) PUT$(cont, key, value)$, to put $value$ under key in container $cont$;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

HCS implements a multi-writer multi-reader key-value storage. HCS is strongly consistent, i.e., it implements atomic [23] (or linearizable [11]) semantics. In distributed storage context, atomicity provides the illusion that a complete operation op is executed instantly at some point in time between its invocation and response, whereas the operations invoked by faulty clients appear either as complete or not invoked at all.

Despite providing strong consistency, HCS is highly available. HCS writes satisfy wait-free [24] semantics, guaranteeing that a write by a correct client eventually completes. On the other hand, HCS reads satisfy marginally weaker finite-write (FW) terminating semantics [25]. Namely, FW-termination guarantees an operation by a correct client to complete always, except when there is an infinite number of writes to the same key. In other words, in HCS, we trade-in read wait-freedom for FW-termination and better performance — namely, guaranteeing read wait-freedom reveals very costly in KVS-based multi-cloud storage systems [8] and significantly impacts storage complexity. We feel that our choice will not be limiting in practice, since FW-termination intuitively offers virtually the same guarantees as wait-freedom for a large number of workloads.

3.3 HCS Protocol

In this section we first briefly give an overview of the HCS protocol. Then 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.



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

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

3.3.1 Overview

At the core of HCS is a multi-writer multi-reader storage protocol. In terms of consistency, the protocol ensures atomic semantics, while in terms of availability it guarantees wait-free write operations and slightly weaker FW-terminating reads.

Storage operations by clients access a collection of cloud storage services, of which up to f can be arbitrary faulty. While HCS consistency is preserved in the face of any number of arbitrary faulty clouds, $f + 1$ correct clouds are needed for availability, amounting to a total of at least $2f + 1$ clouds. The key to shaving off f clouds from the classical $3f + 1$ resilience lower bound is HCS RMDS component. For each key-value pair RMDS maintains corresponding metadata including: (i) a cryptographic hash of the value, (ii) a list of at least $f + 1$ pointers to clouds that store the value and (iii) a timestamp. Besides these critical metadata, to enable various optimizations RMDS also stores the size of value v.

This metadata, despite being lightweight is powerful enough to enable tolerating arbitrary cloud failures, virtually at no additional cost compared to tolerating only crash failures. Intuitively, the cryptographic hash enables end-to-end integrity protection, ensuring that corrupted values are never returned to the application. Since the cryptographic hash is computed by a (trusted) client and never crosses the trust boundary of the private cloud, any unauthorized modification of a value by the public clouds can be detected by means of the original hash.

Furthermore, metadata points to $f + 1$ clouds that have been previously updated, enabling a client to retrieve the correct value despite f of them being arbitrary faulty. In fact, with HCS, as few as $f + 1$ clouds are sufficient to ensure both consistency and availability of read operations (namely GET) — indeed, HCS GET never involves more than $f + 1$ clouds. Additional f clouds (totalling $2f + 1$ clouds) are only needed to guarantee that write operations (namely PUT) are available as well. Note that since f clouds can be faulty, and a value needs to be stored in $f + 1$ clouds for durability, overall $2f + 1$ clouds are required for PUT operations to be available in the presence of f cloud outages.

Finally, besides cryptographic hash and pointers to clouds, metadata includes a timestamp that, roughly speaking, induces a partial order of operations which captures the real-time precedence ordering among operations (atomic consistency). Timestamps are managed by HCS clients and are classical multi-writer timestamps [26], that comprise a monotonically increasing sequence number and clients' id as tiebreaker.[2]

In the following we describe how the HCS protocol works by detailing each storage operation individually.

3.3.2 PUT Protocol

HCS PUT protocol entails a sequence of consecutive steps illustrated in Figure 2. To write a value v under key k, a client first obtains from RMDS the latest authoritative timestamp

[2]We decided against leveraging server-managed timestamps (e.g. provided by ZooKeeper) since we prefer a more generic RMDS implementation to a Zookeeper-specific one. In fact, one of the few primitives that HCS requires from a RMDS implementation is the availability of conditional update operations, supported not only by ZooKeeper but also by a few other NoSQL data stores, such as HBase and MongoDB, which allows reasonably simple porting of RMDS from ZooKeeper to a NoSQL backend.

ts. The client does so by requesting the metadata associated with key k. Timestamp ts is a tuple consisting of a sequence number *seqno* and a client id. The client then computes a new timestamp ts_{new}, whose value is $(seqno + 1, cid)$, where *cid* is the client's identifier. Next, the client combines the key k and timestamp ts_{new} to a new key $k_{new} = k|ts_{new}$ and invokes **put** (k_{new}, v) on $f + 1$ clouds in parallel. Concurrently, the clients starts a timer whose expiration is set to typically observed upload latencies (for a given value size). In the common case, the $f + 1$ clouds reply to the the client in a timely fashion, before the timer expires. Otherwise, the client invokes **put** (k_{new}, v) on up to f secondary clouds (see dashed arrows in Fig. 2). Once the client has received acks from $f + 1$ different clouds, it is assured that the PUT is durable and proceeds to the final stage of the operation.

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

Namely, if the client performs a straightforward update of metadata in RMDS, then it may occur that stored metadata is overwritten by metadata with a lower timestamp (old-new inversion), breaking the timestamp ordering of operations and HCS consistency. To solve the old-new inversions problem, we require RMDS to export an atomic conditional update operation. Then, in the final step of HCS PUT, the client issues conditional update to RMDS which updates the metadata for key k only if the written timestamp ts_{new} is greater than the timestamp for key k that RMDS already stores. In Section 3.4 we describe how we implement this functionality over Apache Zookeeper API; alternatively other NoSQL and SQL DBMSs that support conditional updates can be used.

3.3.3 GET in the common case

HCS GET protocol is illustrated in Figure 2. To read a value stored under key k, the client first obtains from RMDS the latest metadata, comprised of timestamp ts, cryptographic hash h, value size s, as well a list *cloudList* of pointers to $f + 1$ clouds that store the corresponding value. Next, the client selects the first cloud c_1 from *cloudList* and invokes **get** $(k|ts)$ on c_1, where $k|ts$ denotes the key under which the value is stored. Besides requesting the value, the client starts a timer set to the typically observed download latency from c_1 (given the value size s) (for that particular cloud). In the common case, the client is able to download the correct value from the first cloud c_1 in a timely manner, before expiration of its timer. Once it receives value v, the client checks that v hashes to hash h comprised in metadata (i.e., if $H(v) = h$). If the value passes the check, then the client returns the value to the application and the GET completes.

In case the timer expires, or if the value downloaded from the first cloud does not pass the hash check, the client sequentially proceeds to downloading the data from the second cloud from *cloudList* (see dashed arrows in Fig. 2) and so on, until the client exhausts all $f + 1$ clouds from *cloudList*.[3]

[3]In our early prototype, clouds in *cloudList* are ranked by the client by their typical latency in the ascending order, i.e., when reading the client will first read from the "fastest" cloud from *cloudList* and then proceed to slower clouds.

In specific corner cases, caused by concurrent garbage collection (described in Sec. 3.3.4), failures, repeated timeouts (asynchrony), or clouds' inconsistency, the client has to take additional actions in GET. These are described in details in the following, in Section 3.3.5.

3.3.4 Garbage Collection

The purpose of garbage collection is to reclaim storage space by deleting obsolete versions of keys from clouds while allowing read and write operations to execute concurrently.

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. This involves invoking list($k|*$) on every cloud and fetching metadata associated with key k from RMDS. If ts is the latest authoritative timestamp, then for each key k_{old}, where $k_{old} < k|ts$, the client invokes DELETE (k_{old}) on every cloud.

Garbage collection in HCS is performed by the writing client asynchronously in the background. As such, the PUT operation can give back control to the application without waiting for completion of garbage collection.

3.3.5 GET in the worst-case: Consistency Hardening

In the context of cloud storage, there are known issues with weak, e.g., eventual [12] consistency. With eventual consistency, even a correct, non-malicious cloud might deviate from atomic semantics (strong consistency) and return an unexpected value, typically a stale one. In this case, sequential common-case reading from $f + 1$ clouds as described in Section 3.3.3 might not return a value since a hash verification might fail at all $f + 1$ clouds. In addition to the case of inconsistent clouds, this anomaly may also occur if: (i) timers set by the client for a otherwise non-faulty cloud expire prematurely (i.e., in case of asynchrony or network outages), and/or (ii) values read by the client were concurrently garbage collected (Sec. 3.3.4).

To cope with this issues and eventual consistency in particular, HCS introduces consistency hardening. With HCS, we leverage metadata service consistency to mask data inconsistencies in the clouds, effectively allowing availability to be traded off for consistency. This time, however, the tradeoff is under control of the HCS client (e.g., application developer) who can simply require strong consistency from the clouds, as our HCS demonstrates. We describe below our consistency hardening scheme that is appended to HCS common-case GET (as described in Sec. 3.3.3) and executed only if the common-case GET does not return a value.

Roughly speaking, with consistency hardening HCS client indulgently reiterates the GET by reissuing a **get** to all clouds in parallel, and waiting to receive at least one value matching the desired hash. However, due to possible concurrent garbage collection (Sec. 3.3.4), a client needs to make sure it always compares the values received from clouds to the most recent key metadata. This can be achieved in two ways: (i) by simply looping the entire GET including metadata retrieval from RMDS, or (ii) by looping only **get** operations at $f + 1$ clouds while fetching metadata from RMDS only when metadata actually changes.

In HCS, we use the second approach. Notice that this suggests that RMDS must be able

to inform the client proactively about metadata changes. This can be achieved by having a RMDS that supports subscriptions to metadata updates, which is possible to achieve in, e.g.., Apache Zookeeper (using the concepts of watches, see Sec. 3.4 for details). The entire protocol executed upon common-case GET fails (Sec. 3.3.3) proceeds as follows:

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

The complete HCS GET ensures finite-write (FW) termination in presence of eventually consistent clouds. A GET may fail to return a value only theoretically, in case of infinite number of concurrent writes to the same key, in which case the garbage collection at clouds (Sec. 3.3.4) might systematically and indefinitely often remove the written values before the client manages to retrieve them.

3.3.6 DELETE and LIST

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

To delete a value, the client performs the PUT protocol with a special $cloudList$ value $\perp$ denoting the lack of a value. DELETE is local to RMDS and does not access public clouds. Namely, deleting a value creates metadata tombstones in RMDS, i.e. metadata that lacks a corresponding value in cloud storage. Nevertheless, metadata tombstones are necessary for keeping record of the latest authoritative timestamp associated with a given key.

Just like DELETE, LIST operation is local to RMDS. Roughly speaking, our LIST implementation simply retrieves from RMDS all keys associated with a given container $cont$ and filters out deleted keys.

3.3.7 Confidentiality

Adding confidentiality to HCS is straightforward and entails the following modifications to HCS.

During a PUT, just before uploading data to $f+1$ public clouds, the client encrypts the data with a symmetric cryptographic key k_{enc}. Then, in the final step of the PUT protocol

(see Sec. 3.3.2), when the client writes metadata to RMDS using conditional update, the client simply adds k_{enc} to metadata and computes the hash on ciphertext (rather than on cleartext). The rest of the PUT protocol remains unchanged. The client may generate a new key with each new encryption, or fetch the last used key from the metadata service, at the same time it fetches the last used timestamp.

Decryption is also straightforward. Upon fetching metadata from RMDS during a GET, the client also obtains the most recently used encryption key k_{enc}. Then, upon the retrieved ciphertext from some cloud successfully passes the hash test, the client decrypts data using k_{enc}.

3.4 HCS implementation

3.4.1 Overview

We implemented HCS in Java. The implementation pertains solely to the HCS client side since the entire functionality of the metadata service (RMDS) is layered on top of Apache Zookeeper client. Namely, HCS does not entail any modification to the Zookeeper server side. Our HCS client is lightweight and consists of 2030 lines of Java code.

HCS client interactions with public clouds are implemented by wrapping individual native Java SDK clients (drivers) for each particular cloud storage provider[4] into a common lightweight interface that masks the small differences across native client libraries. Initially, our implementation relied on the portable Apache JClouds library [27] which roughly serves the main purpose as our custom wrapper, yet covers dozens of cloud storage providers. However, JClouds introduces its proper performance overhead that prompted us into implementing the cloud driver library wrapper ourselves.

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

3.4.2 RMDS implementation over Zookeeper

In the following, we first briefly recall Zookeeper data model and the Zookeeper API (for more details, please refer to [15]). Then, we detail our RMDS implementation over Zookeeper.

Zookeeper data model and API. Zookeeper data model conveniently maps to that of HCS. Namely, Zookeeper stores data within objects called znodes which are addressed by paths in a hierarchical namespace. Each znode stores some data, which we use to store HCS metadata.

Zookeeper exports a fairly modest API to its applications. The Zookeeper calls relevant to us are: (i) **create**$(path, data)$, which creates znode with path p containing $data$, (ii) **setData**$(p, data)$, which updates an existing znode with path p with $data$, (iii) **getData**(p) to

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

retrieve data stores under znode with p, (iv) **getChildren**(p) (used in HCS LIST) which returns the list of znodes children of the znodes with path p (i.e., all znodes whose path is prefixed by p), and (v) **sync**(), which synchronizes a Zookeeper replica that maintains the client's session with Zookeeper leader and guarantees that reads that follow will be strongly consistent.[5] Finally, Zookeeper allows several operations to be wrapped into a transaction which are then executed atomically.

Besides data, znodes have some specific Zookepeer metadata which should not be confused with HCS metadata which is stored as Zookeeper data. Among this Zookeeper metadata relevant to us is the znode version number vn, that can be supplied as an additional parameter to **setData** operation (i.e., **setData**($p, data, vn$)) which then becomes a conditional update operation which updates znode with path p only if its version number exactly matches vn.

Finally, we use the concept of Zookeeper watches. Watches can be set by several operations, yet we use watches only with **getData**. Watches are clients' subscriptions on znode update notifications; Zookeeper simply triggers a watch at the client when a znode changes — the client is responsible for fetching the actual znode data itself.

Metadata layout. We layout HCS metadata in Zookeeper namespace as follows. For each instance of HCS, we generate a root znode. Then, the metadata pertaining to HCS container $cont$ is stored under Zookeeper path $\langle root \rangle / cont$. In principle, for each HCS key k in container $cont$, we store a znode with path $path_k = \langle root \rangle / cont / k$.

HCS PUT. At the beginning of PUT (k, v), when client fetches the latest timestamp ts for k, we need to make sure that this read from Zookeeper is atomic. To this end, the HCS client issues a **sync**() followed by **getData**($path_k$). This **getData** call returns, besides HCS timestamp ts, the internal version number vn of the znode $path_k$ which the client uses when writing metadata md to RMDS in the final step of PUT.

In the final step of PUT, the client issues **setData**($path_k, md, vn$)) which succeeds only if the znode $path_k$ version is still vn. If the Zookeeper version of $path_p$ changed, the client retrieves the new authoritative HCS 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 guarantees wait freedom of PUT since only a finite number of concurrent PUT operations can use a timestamp smaller than ts.

HCS GET. In interacting with RMDS during GET, HCS client simply needs to make sure its metadata is read atomically. To this end, a client always issues a **sync**() followed by **getData**($path_k$), just like in our PUT protocol.

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

3.4.3 Optimizations

Cloud latency ranks. In our HCS implementation, clients rank clouds by latency and "prefer" clouds with lower latency. HCS client then uses these cloud latency ranks in common case to: (i) write to $f + 1$ clouds with the lowest latency in PUT, and (ii) to select from *cloudList* the cloud with the lowest latency as preferred cloud in GET.

Initially, we implemented the cloud latency ranks by reading once (i.e., upon initialization of the HCS client) a default, fixed-size (100kB) object from each of the public clouds. Interestingly, during our experiments, we observed that the cloud latency rank significantly varies with object size as well as the type of the operation (PUT vs. GET). Hence, our implementation establishes several cloud latency ranks depending on the file size and the type of operation. In particular, we establish four latency ranks: one for each of reading and writing of small and medium files (up to 10MB in size), as well as additional two for reading (resp., writing) objects larger than 10MB. These cloud latency ranks are established by executing, upon initialization of the client: (i) GET (resp., PUT) of a 100kB object for small/medium-sized files, and (ii) GET (resp., PUT) of a 10MB file for large-size. In addition, HCS client can be instructed to refresh these latency ranks when necessary.

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

Caching. Our HCS implementation enables data caching on private portion of the system. HCS design enables simple integration with different caching policies. We implemented simple write-through cache and caching-on-read policies. With write-through caching enabled, HCS client simply writes to cache in parallel to writing to clouds. On the other hand, with caching-on-read enabled, HCS client upon returning a GET value to the application, writes lazily the GET value to the cache. In our implementation, we use memcached distributed cache that exports a key-value interface just like public clouds. Hence, all HCS writes to the cache use exactly the same addressing as writes to public clouds (i.e., using **put**$(k|ts, v)$).

Finally, to leverage cache within a GET, HCS client upon fetching metadata always tries first to read data from the cache (i.e., by issuing **get** $(k|ts)$ to memcached). Only in the case of a cache miss, HCS client proceeds normally with a GET, as described in Sections 3.3.3 and 3.3.5.

3.5 HCS extensions in Year 2

In this section we briefly discuss two possible extensions of HCS that we plan to explore in Year 2 of Cloudspaces.

Alternative deployments of metadata service. HCS RMDS runs on trusted resources and hence can rely on crash-tolerant replication protocol that underlies Apache Zookeeper to handle metadata server crashes. Hence, HCS needs $2t + 1$ metadata (Zookeeper) servers to tolerate t faulty ones. Alternatively, a system with HCS architecture could be deployed in entirely untrusted environment. However in this case, we would need a "BFT Zookeeper"; i.e., a BFT state machine replication protocol (e.g., PBFT [28]) with Zookeeper API on top of such replication protocol. Such protocol would require $3t + 1$ metadata servers to tolerate t malicious ones [29]. Yet, such a system would still employ only $2f + 1$ public clouds (i.e., untrusted data repositories) to mask f malicious ones. A possible choice for such a deployment could be a collocation, i.e., to leverage $3t + 1$ virtual machines in the clouds, all acting as metadata replicas, but only $2t + 1$ of those acting as data replicas. In other words, such alternative HCS design would respect the classical $3t + 1$ lower bound on the number of replicas in BFT storage [30], yet only $2t + 1$ of those would be used to store data.

In addition, RMDS could be deployed over wide-area networks to facilitate remote accesses from different geographical locations.

Erasure coding. In this paper, we focus on using data replication in HCS. To reduce storage blowup associated with replication, some multi-cloud storage systems employ erasure coding, e.g., Depsky in its Depsky-CA and Depsky-E variants [7]. For example, Depsky-E employs $3f + 1$ clouds, stores data to $2f + 1$ clouds in the common case and reads from $f + 1$ clouds using erasure coding — this results in $\frac{2f+1}{f+1}$ storage blowup. In the typical case where $f = 1$, Depsky-E has storage blowup of 1.5 whereas our replication-based HCS yields blowup equal to 2.[6]

This evokes back the classical arguments that contrast replication and erasure-coding [31] and their tradeoffs in high-availability vs. storage blowup. While we believe that exploring these classical tradeoffs in the game-changing context of cloud storage requires deeper insight we make one important observation.

Namely, to guarantee availability our replication-based HCS requires $n = 2f + 1$ public clouds in the worst case. However, given the abundance of cloud providers and their pay-as-you-go model, it might make sense to use more cloud providers in combination with erasure coding to reduce storage blowup. A back of the envelope calculation suggests that with up to f malicious clouds and n clouds in total, using erasure coding one could maintain HCS semantics while achieving the blowup of $\frac{n-f}{n-2f}$. Since nothing prevents $n >> f$ (e.g., using 10 or more clouds to mask 1 possibly faulty cloud), for large values of n this blowup tends to 1 and is clearly optimal. Our system architecture does not prevent such erasure coding variants of HCS that, however induce numerous tradeoffs and that we plan to address in Year 2 towards Deliverable 3.2.

[6] Notice that 1.5 is Depsky-E blowup in case of static data, i.e., write-once data that does not change. In case of dynamic and shared data, Depsky blowup is considerably higher since Depsky requires storing the entire history of the data object, unlike HCS.

4 Early prototype of the Adaptive edge platform

During the first year we have developed an early prototype of the Adaptive Edge platform that is completely integrated with the open source StackSync Personal Cloud. We focused our efforts in the transparent integration of Bittorrent edge technologies with StackSync and OpenStack. Note that this prototype has been presented as a demonstration paper in the IEEE P2P'13 conference [32].

Content delivery in current cloud providers make use of HTTP as a transfer protocol, missing the opportunity of offloading storage servers from doing much of the serving when multiple clients download the same content. Besides, bandwidth costs increase with the number of users a cloud provider serves. Making a smarter, efficient and effective use of the otherwise limited available bandwidth might encourage the proliferation of a higher number of competing personal clouds.

A good opportunity to overcome the high cost of data transfers is to replace regular HTTP unicast transfers by BitTorrent [1]. Since its conception, BitTorrent has proven to be one of the most effective techniques for distributing large content. Rather than directly downloading a file from the server, the BitTorrent protocol allows clients to join a "swarm" of hosts to download and upload portions of the file from each other simultaneously. Because BitTorrent utilizes the upload bandwidth of clients to offload the original content source, the result is that content distribution becomes less bandwidth intensive for the server, saving bandwidth costs.

Unfortunately, the adaptation of BitTorrent to fit the requirements of personal storage services is an open issue. While BitTorrent has been adopted by market leaders like Amazon to reduce the network costs of the S3 service [33], the integration of BitTorrent into the operational cycle of a personal cloud is not trivial. This is mainly because BitTorrent is meant to complement, not replace, regular file synchronization in scenarios where there is a simultaneous demand for the same set of objects. This means that BitTorrent cannot run as a standalone service within the personal cloud but rather must operate as an integral part of the system.

In this section, we briefly discuss the experiences and insights gained from integrating BitTorrent into StackSync.

4.1 Goals

The adoption of BitTorrent aims at keeping the general features expected in any personal cloud like file synchronization and data sharing but overcoming one important limitation: *the data transfer bottleneck* [34]. However, contrarily to the Amazon S3 service where it must be explicitly requested the .torrent file associated with an object, the use of BitTorrent in a personal cloud should remain *oblivious* to users to not affect usability. Typically, personal clouds like `Dropbox` synchronize files dynamically with limited or no human intervention. And clearly, asking the users to manually start a BitTorrent client would impair the usability of the whole system. This implies that BitTorrent cannot run as a standalone service. Rather, it must be activated by the personal cloud service whenever it seems advantageous to do so and without the involvement of users. Achieving this transparency requires the introduction

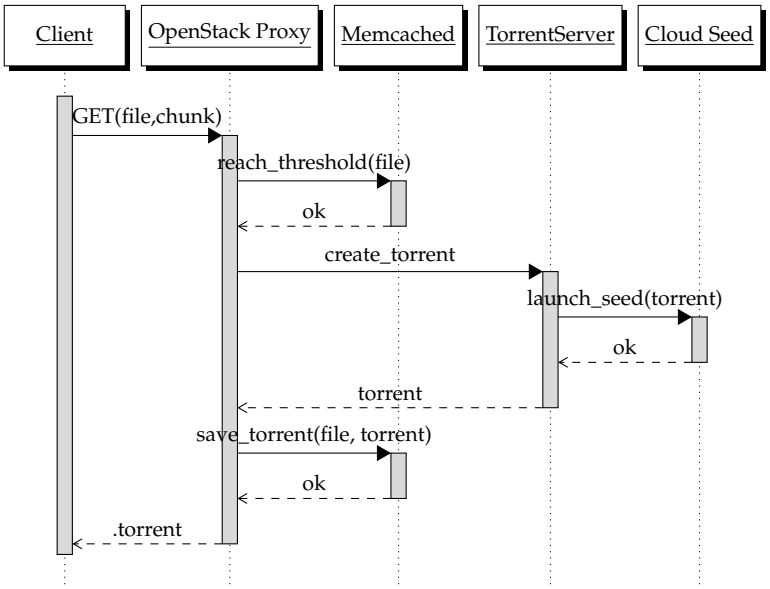


Figure 3: Interaction of a client with the cloud to switch to BitTorrent.

of important changes in the "classical" behaviour of BitTorrent as discussed below. The first obvious change is that the peers within a swarm will no longer be independent BitTorrent clients but personal cloud clients with a BitTorrent client embedded into it.

Very succinctly, the idea is that the personal cloud service (Mediator) monitors user activity and upon detecting a certain critical mass of users, it generates a .torrent file "on the fly" to offload the servers in the data center from doing much of the serving. In this case, the data center will act as a seed and the coordinator will instrument the personal cloud clients to optimize content distribution and bandwidth consumption. To the best of our knowledge, no such a novel use of BitTorrent has been reported to date in the literature.

4.2 Operation

To better understand how the components interact with one another during a synchronization operation, Fig. 3 shows the interactions between one client and the Swift proxy in the cloud. The interactions are the following:

1. Clients use the standard Swift REST API to ask the server for the missing and the modified portions of files since the last synchronization. The target file is added as metadata to the HTTP requests, so that the Swift proxy can check whether a sufficient number of clients are interested in the file. To maintain the popularity counter for each file, we used memcached. Memcached is a free open source in-memory key-value store for caching small pieces of data. The use of memcached served two purposes: to reduce the number of calls to the database and to enable inter-proxy communication when several proxies coexist for scalability reasons.

2. Whenever the threshold on the number of users is reached, the proxy requests the `TorrentServer` the creation of the .torrent file and the bootstrap of the new swarm.

3. The `TorrentServer` in turn launches a BitTorrent seed in the data center that immediately joins the swarm to start serving pieces to the clients.

4. In addition, the proxy stores the .torrent file associated with the file in the memcached servers while the swarm is active.

5. Finally, the Swift proxy responds to the clients with the .torrent file and the instruction to change to BitTorrent. Upon reception of the response, the clients join the swarm using their local BitTorrent instance. From this point onward, new clients that request pieces of this file will obtain from the proxy the already existing .torrent file and join the existing swarm.

4.3 Current Challenges

Although in early stages of prototyping, our implementation is ready for testing in a scenario with real users. However, this solution opens the door to new challenges which remain to be studied more in depth.

One of the key issues we faced was the determination of the suitable moment to initiate BitTorrent and pause the HTTP protocol for content delivery. It is well known that until the system has a critical mass of concurrent users, peer-to-peer serving in BitTorrent is not effective [35]. For this reason, we based the switching decision on the number of simultaneous clients that are downloading a given file. Concretely, the switch to BitTorrent occurs when the number of concurrent clients surpasses a certain threshold, namely `MAX_USER`. Although this approach seems a simple heuristic to follow given our experience with BitTorrent, potential alternatives should be considered and evaluated. For example, a natural path to follow would be to also consider the contributed bandwidth of users as our threshold to switch from HTTP to a swarm based distribution of content. We are currently developing and evaluating a variety of adapter pluggable policies for HTTP to BT switching and vice versa.

Another key aspect that needs further investigation is the bandwidth allocation in a multi-swarm scenario. Since the outgoing bandwidth in a data center fabric is finite, one important question is how to achieve the shortest possible download times across multiple swarms given a bandwidth budget. Finding the optimal allocation of bandwidth among the managed swarms is not trivial, because requires developing heuristics that take into account factors such as the file size, the size of swarms and the nature of clients that make up each swarm. This is clearly an issue that we are currently evaluating as could potentially save a vast amount of bandwidth resources on the data center side at no perceptible cost to the end-user.

4.4 Deployment and management of the Adaptive cloud platform testbed

In this section we explain the different efforts to deploy a massive testbed for the first user trials and distributed testing experiments. As explained in D2.2, two partners deployed cloud platform testbeds for the CloudSpaces project: TISSAT and URV. Both testbeds include a complete OpenStack Swift installation (version 1.8.0) enabling massive real user tests in

this platform. Furthermore, both platforms include the last version of StackSync adaptive server components.

To test the platform in real settings a number of components have been deployed like: software update, log server, monitoring service and an integrated bittorrent service.

4.4.1 Log server

As StackSync is not a final release, it is necessary to obtain as much information as possible to solve bugs and improve its performance. For this reason it has been developed a log server that will receive, process and store the errors that StackSync clients throw.

Each time that the client has an exception, it will send a report to the server using a well defined REST API. The request body contains the information in plain text but compressed to save as much bandwidth as possible. The information is stored in a StackSync log folder in the server-side.

Periodically, another program which runs in the server will process the logs. First of all, it will decompress them and parse all the information in different fields (time, thread that throws the exception and filename, among others). After that, it will map this information in a PostgreSQL database to have the information accessible to a further review. Finally, once the log is processed, it is moved into another folder.

If, for any reason, it is impossible for the client to send the reports, the client will try to send them later.

All the information that is used does not contain user personal information. It contains the necessary information for the developer to fix the bug.

4.4.2 Software update

As a complement of the log server, we have developed a software updater for the desktop clients. The aim is to solve bugs using the clients reports and release new versions of StackSync without them using this software.

All the update process will be transparent for the user. To achieve this, StackSync has a thread running in background that checks periodically if there is a newer version ready. In the current implementation, it is pulling the server using a REST API, but in a future it could be improved using a push strategy and a notification system such as ObjectMQ (see D2.2).

When a new version is released, it is downloaded from the server and the old one is replaced. With this method we ensure that all the clients are running the last version of our software.

4.4.3 Monitoring service

Nowadays it is almost impossible to obtain real traces from private Personals Clouds, private companies do not make this information public. Even the companies involved in the

project (Canonical and eyeOS) had legal and technical problems to share their traces with the rest of partners. For this reason we have created a monitoring service that will create traces of our StackSync system.

The fact that real users are using StackSync, allows us to create real traces. These traces will help us to understand users behaviours and possible patterns, which will be very interesting for a future use in research.

All these traces are generated in the server-side without disclosing personal user information of any type. For example, instead of using a username it will be used a random identifier which will identify this user in the system. The traces follow the format specified in the deliverable D2.2.

In the first year of the project small tests have been performed in both testbeds. But as planned in the project, the first open trials for early adopters will begin in October/November 2013. We aim to create as soon as possible a user community that may provide feedback for the CloudSpaces project.

4.5 Proximity-aware services

We outline two main scenarios where Personal Clouds can benefit from proximity services: (i) performance improvements in content distribution and (ii) discovery of close-by services.

In the first scenario, synchronization clients can reduce their traffic with Cloud storage services if the content can be obtained from close-by clients. The classical example is the LAN sync protocol included in Dropbox that connects directly clients in the same LAN. The protocol accelerates synchronization traffic thanks to fast local connectivity while also reducing communication with the remote cloud services.

Another promising example for proximity-aware performance improvements can be the integration of P2P protocols with Personal Cloud services to benefit from the bandwidth resources of peers. In this case, proximity can be considered in terms of network proximity or locality. A good example of this trend is the adaptive content distribution technique presented in the previous section. In this project, we are integrating BitTorrent technologies with Personal Clouds to reduce bandwidth consumption in the data center.

An important insight is that this kind of proximity-aware network aware protocols are difficult to be standardized with APIs that may be widely adopted. Since synchronization protocols of different providers are quite heterogeneous and complex, it is very difficult that they can interoperate at this level. For this reason, we will not devote resources to propose any standard API for proximity-aware synchronization services.

Another candidate scenario for proximity aware services is the discovery of close-by spaces. For example, attendants to a conference could discover the conference space created to promote the collaboration between participants. Another use case could be the interaction of clients in a shopping mall with the near-by promotional spaces of the different shops. Clients could obtain discounts or even become permanent members of some shopping spaces to obtain benefits. These scenarios open new ways of integrating digital spaces (Personal Clouds) with physical locations and objects.

In order to discover close-by spaces there a a number of alternatives and protocols that may be used. From Web Geolocation APIs, to device GPS location APIs, and even the use of physical tokens to grant controlled access to a space. For example, it is possible to use technologies such as NFC (Near Field Communication) to create physical tokens granting access to specific spaces.

But again, it is beyond the scope of this project to standardize a novel location-based or discovery technology or API. On the contrary, our efforts on storage and interoperable sharing could then be used in a myriad of location-based scenarios and contexts. As a conclusion, regarding proximity services, we will focus in this project in network proximity improvements thanks to the use of edge resources.

5 Comparison to related work

Multi-cloud storage systems. Several storage systems (e.g., [36, 37, 38, 7, 8]) have used multiple clouds in boosting data robustness, notably reliability and availability. Early multi-cloud systems such as RACS [36] and HAIL [37] assumed immutable data, hence not addressing any concurrency aspects.

Multi-cloud storage systems closest to HCS are Depsky [7] and ICStore [8]. ICStore is a robust cloud storage system that models cloud faults as outages and implements wait-free access to shared data. HCS advantages over ICStore include tolerating malicious clouds and smaller storage blowup[7] while paying a small price in FW-terminating reads (as opposed to ICStore's wait-free reads). On the other hand, Depsky considers malicious clouds, yet requires $3f + 1$ clouds, unlike HCS. Furthermore, Depsky consistency and availability guarantees are weaker than those of HCS, even when clouds behave as strongly consistent. As we already discussed, the distinctive feature of HCS is consistency hardening which guarantees HCS atomicity even in presence of eventually consistent clouds, which may harm the consistency guarantees of both ICStore and Depsky.

Finally, a recent addition to multi-cloud storage systems is SPANStore [38] which seeks to minimize the cost of use of multi-cloud storage. However, the reliability guarantees of SPANStore are considerably below those of HCS. Namely, SPANStore is not robust, as it features a centralized cloud placement manager component which is a single point of of failure. In addition, SPANStore considers crash-only clouds and uses leased lock-based writes, which yield obstruction-free [39] availability at best.

Separating data from metadata. Separating metadata from data is not a novel idea in distributed systems. For example, in the Hadoop Distributed File System (HDFS), modeled after the Google File System [40], HDFS NameNode is responsible for maintaining metadata, while data is stored on HDFS DataNodes. However, the idea of metadata/data separation particularly excels in our contexts of cloud storage (allowing consistency hardening in HCS) and in untrusted storage in general.

Namely, in the untrusted storage context, separating metadata from data allows HCS to tolerate f malicious data repositories using only $2f + 1$ data storage repositories. This is a capital benefit since all prior robust Byzantine fault tolerant (BFT) storage systems have used $3f + 1$ data repositories to mask f malicious ones, which made the deployment of BFT storage systems very expensive. Here, it is very important to notice that, with metadata/data separation, the cost related to BFT data replication becomes equal to the cost of crash-tolerant data replication.[8] Indeed, crash-tolerant storage protocols similar to HCS, that separate metadata from data (e.g., Gnothi [41]), need $2f + 1$ data repositories nevertheless, and this can be shown optimal.

In a sense, HCS extends the idea of separating control and data planes in BFT systems, first introduced in [42] in the context of replicated state machines (RSM), to storage. While the RSM approach of [42] could obviously be used for implementing storage as well, HCS

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

[8] As discussed below, the cost of BFT metadata replication remains higher as mandated by the lower bound of [30].

proposes a far more scalable and practical solution, while also tolerating pure asynchrony across data communication links, unlike [42].

Other notable storage systems that separate metadata from data include LDR [43] and BookKeeper [44]. LDR [43] implements asynchronous multi-writer multi-reader read/write storage and, like HCS, uses pointers to data storage nodes within its metadata and requires $2t + 1$ data storage nodes. However, unlike HCS, LDR considers full-fledged servers as data storage nodes and tolerates only their crash faults. BookKeeper [44] implements reliable single-writer multi-reader shared storage for logs. BookKeeper stores metadata on Zookeeper servers (bookies) and data (i.e., log entries) in log files (ledgers). Like in HCS RMDS, bookies point to ledgers, facilitating writes to $f + 1$ ledgers and reads from a single ledger in common-case. However, HCS differs significantly from BookKeeper: namely, HCS supports multiple writers, tolerates malicious faults of data repositories and is designed with different deployment environment and applications in mind.

Systems based on trusted components. Several systems have used trusted hardware components to reduce the overhead of BFT replication to $2t + 1$ replicas, typically in the context of RSM (e.g., [45, 46, 47, 48]). Some of these systems, like CheapBFT [47], employ only $t + 1$ replicas in the common case.

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

Integrating BitTorrent in the Data center. A number of studies have tried to combine BitTorrent content distribution technologies with Cloud environments. In particular, the efficiency of the BitTorrent protocol makes it especially suitable for massive content distribution while reducing bandwidth costs in the Cloud. A prominent example is Amazon's standard offering for BitTorrent content distribution in the Amazon S3 Storage service. Apart from the standard REST and SOAP APIs, it is possible to retrieve objects stored in S3 using BitTorrent. Users must explicitly request for a torrent file of a content by appending ?torrent to the GET request.

In [49], authors evaluate the use of Amazon S3 services for Science Grids. They pay special attention to the use of BitTorrent in S3 as a cooperative cache that may reduce costs when transferring large amounts of data. Their experiments with an Amazon S3 seed and several external seeds show that S3 contributes a large percentage of the data volume (around 50

Many previous works have focused on reducing download times for large contents using BitTorrent in Cloud settings. Either for bulk synchronous content distribution or for reducing transfer times for cloud virtual images [50] [51] BitTorrent proved to be a very efficient technology that outperforms classical centralized transfer solutions. For example in [52] authors demonstrate that their BitTorrent-based solution for distributing virtual machines delivers up to an 30x speedup over traditional remote file system approaches.

Another related work is [53] where authors present a content distribution system based on managed swarms. They use a centralized coordinator to monitor the activity of swarms and assign bandwidth to peers in order to optimize the content distribution. This work is very related with the problem of bandwidth allocation in a multi-swarm scenario that we want to address. A clear difference is that we aim to reduce bandwidth consumption in the data center in scenarios of restricted output bandwidth.

Our approach is novel because it provides transparent and adaptive mechanisms to switch between traditional HTTP and BitTorrent technologies in Cloud Storage environments. None of the previous approaches target this transparency or adaptivity. Furthermore, we will demonstrate how our adaptive policies can efficiently handle flash crowds in OpenStack while reducing transfer costs inside the Cloud.

References

[1] B. Cohen, "Incentives build robustness in bittorrent," in Proc. of 1st Workshop on Economics of Peer-to-Peer Systems, 2003.

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

[3] "EMC: Transform to a Hybrid Cloud," http://www.emc.com/campaign/global/hybridcloud/index.htm.

[4] "IBM Hybrid Cloud Solution," http://www-01.ibm.com/software/tivoli/products/hybrid-cloud/.

[5] "Egnyte: Hybrid Cloud for NetApp," http://www.egnyte.com/hybrid-cloud/hybrid-cloud-for-netapp.html.

[6] "Microsoft lures punters to hybrid storage cloud with free storage arrays," http://www.theregister.co.uk/2013/09/24/microsoft_lures_punters_to_hybrid_storage_cloud_with_free_storage_arrays/.

[7] A. N. Bessani, M. P. Correia, B. Quaresma, F. André, and P. Sousa, "Depsky: dependable and secure storage in a cloud-of-clouds," in EuroSys, 2011, pp. 31–46.

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

[9] Z. Wu, M. Butkiewicz, D. Perkins, E. Katz-Bassett, and H. V. Madhyastha, "Cspan: cost-effective geo-replicated storage spanning multiple cloud services," in SIGCOMM, 2013, pp. 545–546.

[10] C. Cachin, R. Haas, and M. Vukolić, "Dependable storage in the Intercloud," IBM Research, Tech. Rep. RZ 3783, 2010.

[11] M. P. Herlihy and J. M. Wing, "Linearizability: A Correctness Condition for Concurrent Objects," ACM Trans. Program. Lang. Syst., vol. 12, no. 3, 1990.

[12] W. Vogels, "Eventually consistent," Commun. ACM, vol. 52, no. 1, pp. 40–44, 2009.

[13] P. Bailis and A. Ghodsi, "Eventual consistency today: limitations, extensions, and beyond," Commun. ACM, vol. 56, no. 5, pp. 55–63, 2013.

[14] "Apache Zookeeper," http://zookeeper.apache.org/.

[15] P. Hunt, M. Konar, F. P. Junqueira, and B. Reed, "Zookeeper: wait-free coordination for internet-scale systems," in Proceedings of the 2010 USENIX conference on USENIX annual technical conference, ser. USENIX ATC'10. Berkeley, CA, USA: USENIX Association, 2010, pp. 11–11.

[16] "memcached," http://memcached.org/.

[17] "Amazon S3," http://aws.amazon.com/s3/.

[18] "OpenStack Swift," http://docs.openstack.org/developer/swift/.

[19] L. Lamport, "The part-time parliament," ACM Trans. Comput. Syst., vol. 16, no. 2, pp. 133–169, 1998.

[20] P. M. Thambidurai and Y.-K. Park, "Interactive consistency with multiple failure modes," in SRDS, 1988, pp. 93–100.

[21] M. Pease, R. Shostak, and L. Lamport, "Reaching Agreement in the Presence of Faults," J. ACM, vol. 27, no. 2, 1980.

[22] C. Dwork, N. Lynch, and L. Stockmeyer, "Consensus in the presence of partial synchrony," J. ACM, vol. 35, no. 2, pp. 288–323, Apr. 1988.

[23] L. Lamport, "On Interprocess Communication." Distributed Computing, vol. 1, no. 2, pp. 77–101, 1986.

[24] M. Herlihy, "Wait-Free Synchronization," ACM Trans. Program. Lang. Syst., vol. 13, no. 1, 1991.

[25] I. Abraham, G. Chockler, I. Keidar, and D. Malkhi, "Byzantine Disk Paxos: Optimal Resilience with Byzantine Shared Memory," Distributed Computing, vol. 18, no. 5, pp. 387–408, 2006.

[26] N. A. Lynch and A. A. Shvartsman, "RAMBO: A Reconfigurable Atomic Memory Service for Dynamic Networks," in Proceedings of DISC, 2002, pp. 173–190.

[27] "Apache JClouds," http://jclouds.incubator.apache.org/.

[28] M. Castro and B. Liskov, "Practical Byzantine fault tolerance and proactive recovery," ACM Trans. Comput. Syst., vol. 20, no. 4, pp. 398–461, 2002.

[29] L. Lamport, R. E. Shostak, and M. C. Pease, "The byzantine generals problem," ACM Trans. Program. Lang. Syst., vol. 4, no. 3, pp. 382–401, 1982.

[30] J.-P. Martin, L. Alvisi, and M. Dahlin, "Minimal Byzantine Storage," in Proceedings of DISC, 2002, pp. 311–325.

[31] R. Rodrigues and B. Liskov, "High availability in DHTs: Erasure coding vs. replication," in IPTPS, 2005, pp. 226–239.

[32] R. Chaabouni, P. G. Lopez, M. S. Artigas, S. F. Celma, and C. Cebrian., "Boosting content delivery with bittorrent in online cloud storage services," in Proceedings of IEEE P2P 2013, 2013.

[33] M. R. Palankar, A. Iamnitchi, M. Ripeanu, and S. Garfinkel, "Amazon s3 for science grids: a viable solution?" in Proc. of DADC'08, 2008, pp. 55–64.

[34] M. Armbrust, A. Fox, R. Griffith, A. D. Joseph, R. Katz, A. Konwinski, G. Lee, D. Patterson, A. Rabkin, I. Stoica, and M. Zaharia, "A view of cloud computing," Commun. ACM, vol. 53, no. 4, pp. 50–58, 2010.

[35] A. Bharambe, C. Herley, and V. Padmanabhan, "Analyzing and improving a bittorrent networks performance mechanisms," in Proc. of IEEE INFOCOM 2006, 2006, pp. 1–12.

[36] H. Abu-Libdeh, L. Princehouse, and H. Weatherspoon, "RACS: a case for cloud storage diversity," in SoCC, 2010, pp. 229–240.

[37] K. D. Bowers, A. Juels, and A. Oprea, "HAIL: a high-availability and integrity layer for cloud storage," in ACM Conference on Computer and Communications Security, 2009, pp. 187–198.

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

[39] M. Herlihy, V. Luchangco, and M. Moir, "Obstruction-Free Synchronization: Double-Ended Queues as an Example," in Proceedings of ICDCS, 2003.

[40] S. Ghemawat, H. Gobioff, and S.-T. Leung, "The google file system," in SOSP, 2003, pp. 29–43.

[41] Y. Wang, L. Alvisi, and M. Dahlin, "Gnothi: separating data and metadata for efficient and available storage replication," in Proceedings of the 2012 USENIX conference on Annual Technical Conference, ser. USENIX ATC'12. Berkeley, CA, USA: USENIX Association, 2012, pp. 38–38.

[42] J. Yin, J.-P. Martin, A. Venkataramani, L. Alvisi, and M. Dahlin, "Separating agreement from execution for byzantine fault tolerant services," in SOSP, 2003, pp. 253–267.

[43] R. Fan and N. Lynch, "Efficient Replication of Large Data Objects," in Proceedings of DISC, 2003, pp. 75–91.

[44] F. P. Junqueira, I. Kelly, and B. Reed, "Durability with bookkeeper," Operating Systems Review, vol. 47, no. 1, pp. 9–15, 2013.

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

[46] B.-G. Chun, P. Maniatis, S. Shenker, and J. Kubiatowicz, "Attested append-only memory: making adversaries stick to their word," in SOSP, 2007, pp. 189–204.

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

[48] G. S. Veronese, M. Correia, A. N. Bessani, L. C. Lung, and P. Veríssimo, "Efficient byzantine fault-tolerance," IEEE Trans. Computers, vol. 62, no. 1, pp. 16–30, 2013.

[49] M. R. Palankar, A. Iamnitchi, M. Ripeanu, and S. Garfinkel, "Amazon s3 for science grids: a viable solution?" in Proceedings of the 2008 international workshop on Data-aware distributed computing. ACM, 2008, pp. 55–64.

[50] R. Wartel, T. Cass, B. Moreira, E. Roche, M. Guijarro, S. Goasguen, and U. Schwickerath, "Image distribution mechanisms in large scale cloud providers," in Cloud Computing Technology and Science (CloudCom), 2010 IEEE Second International Conference on. IEEE, 2010, pp. 112–117.

[51] M. Schmidt, N. Fallenbeck, M. Smith, and B. Freisleben, "Efficient distribution of virtual machines for cloud computing," in Parallel, Distributed and Network-Based Processing (PDP), 2010 18th Euromicro International Conference on. IEEE, 2010, pp. 567–574.

[52] J. Reich, O. Laadan, E. Brosh, A. Sherman, V. Misra, J. Nieh, and D. Rubenstein, "Vmtorrent: virtual appliances on-demand," ACM SIGCOMM Computer Communication Review, vol. 40, no. 4, pp. 453–454, 2010.

[53] R. Peterson and E. G. Sirer, "Antfarm: Efficient content distribution with managed swarms." in NSDI, vol. 9, 2009, pp. 107–122.