SEVENTH FRAMEWORK PROGRAMME

CloudSpaces

(FP7-ICT-2011-8)

Open Service Platform for the Next Generation of Personal Clouds

D3.3 Final results and software release

Due date of deliverable: 30-09-2015
Actual submission date: 06-10-2015

Start date of project: 01-10-2012 Duration: 33 months

Summary of the document

Document Type	Deliverable
Dissemination level	Public
State	Final
Number of pages	50
WP/Task related to this document	WP3
WP/Task responsible	EUR
Author(s)	Paolo Viotti (EUR), Rahma Chaabouni (URV)
Partner(s) Contributing	EUR, URV
Document ID	CLOUDSPACES_D3.3_150701_Public.pdf
Abstract	This document is a report on the software developed and final results obtained through the research carried out in the context of the CloudSpaces project. First it describes AWE (Asynchronous Wait-free Erasure-coded storage protocol), a novel approach to solve the issues related to erasure coded storage over distributed and unstrusted stores. Along this line of research on practical solutions to cloud storage inherent issues, we also present Hybris. Hybris a multi-cloud storage library we developed and released, both as stand-alone library and as backend of the StackSync prototype Finally, this report presents also our proposal for the adaptive management of the cloud's bandwidth resources. The key idea is to use a peer-to-peer protocol (BitTorrent) instead of HTTP when the load on the cloud becomes high. This approach can be easily integrated with StackSync and two algorithms are proposed to manage the download protocols and to define the bandwidth allocation strategy.
Keywords	Cloud storage, fault tolerance, consistency, file sharing, interoperability, Personal Cloud

Table of Contents

1 Executive summary

This document is a report on the software developed and final results obtained through the research carried out in the context of the Cloudspaces project.

First, it describes AWE (Asynchronous Wait-free Erasure-coded storage protocol), a novel approach to solve the issues related to erasure coded storage over distributed and unstrusted stores. Along this line of research on practical solutions cloud storage inherent issues, we also present Hybris. Hybris is a multi-cloud storage library we developed and released[1], in conjunction with its integration as backend of the StackSync prototype[2].

Finally, we present also our proposal for the adaptive management of the cloud's bandwidth resources. The key idea is to use a peer-to-peer protocol (BitTorrent) instead of HTTP when the load on the cloud becomes high. This approach can be easily integrated with StackSync and two algorithms are proposed to manage the download protocols and to define the bandwidth allocation strategy.

[1] see: `https://github.com/pviotti/hybris`

[2] see `https://github.com/pviotti/stacksync-desktop`

2 Introduction

In this deliverable we provide a practical and detailed description about some technical results obtained by the research of the different contributors of the CloudSpaces project.

We will analyse the use of erasure coding (to minimize the storage consumption), in a solution to handle untrusted and heterogeneous cloud repositories. Further, we will tackle the problem of untrusted storage from another perspective: the challenging integration between private and public clouds, and the related trade-offs on consistency and fault tolerance.

Namely, we present:

- AWE , the first erasure-coded distributed implementation of a multi-writer multi-reader read/write storage object that is, at the same time: (1) asynchronous, (2) wait-free, (3) atomic, (4) amnesic, (i.e., where nodes store a bounded number of values), and (5) Byzantine fault-tolerant (BFT), i.e., tolerating untrusted nodes, using the optimal number of nodes. AWE maintains metadata separately from bulk data, which is encoded into fragments with a k-out-of-n erasure code and stored on dedicated data nodes that support only simple reads and writes. Furthermore, AWE is the first BFT storage protocol that uses only $n = 2t + k$ data nodes to tolerate t Byzantine faults, for any $k \geq 1$. AWE is efficient and uses only lightweight cryptographic hash functions. We present further details of AWE in Section 3.
- Hybris, an advanced version of our initial system described in D3.1, augmented with support for transactional writes, tunable consistency and erasure coding, following the guidelines behind AWE. Hybris is a multi-cloud storage backend that orchestrates heterogeneous public clouds. It provides a robust and efficient storage abstraction over multiple clouds that can be used as Personal Cloud backend in CloudSpaces. Prototyped and designed by EUR, Hybris is described in Section 4.

In Section 5, we will present another case of study on cloud storage. While AWE and Hybris approached cloud storage issues from the users' point of view, this work focuses on the techniques to exploit peer-to-peer technologies to offload cloud providers and ease files distribution. These contributions can be summarized as follows:

- Comparative study between BitTorrent and HTTP: It is commonly believed that BitTorrent is not convenient for the distribution of small files. But, to our knowledge, there is no proof of such assumption. Wei et al. noticed in [1] that, in their specific experimental settings, BitTorrent outperforms the FTP protocol only when the file size is greater than 20 MB. However, in practice, we found that BitTorrent can be efficient for small files [2] which confutes the general statement that BitTorrent is not effective for small files based on a real experimental study.
  In [2], we propose an analytic estimation of the distribution time in BitTorrent that takes into account the overheads related to the nature of the protocol. In addition, we introduce two general metrics to decide when it is better to use one protocol with respect to the other: the gain and the offload ratios. The gain measures the degree of improvement in download time of BitTorrent relative to HTTP. The offload ratio quantifies the amount of data that can be offloaded if the peers adopt BitTorrent. We

validate in [2] all the proposed formulas with focus on small files.

- Switching algorithm: We propose a dynamic algorithm which uses simple parameters that can be collected by the system and evaluates the efficacy of using HTTP and BitTorrent as a distribution protocol for each requested file. Based on the load of the seed and the predefined switching constraints, the algorithms decides the most suitable protocol for each case and provides the corresponding bandwidth allocations at the swarm level.
  The algorithm is validated later using a real trace of the Ubuntu One System [3]. We measure the amount of data that can be offloaded based on different time constraints. We notice that the overall offloaded data volume exceeds 16% of the total amount of data exchanged. From an economic point of view, this corresponds to savings of the order of hundreds to thousands of dollars per month.

- Bandwidth allocation algorithm: The bandwidth allocation algorithm aims to minimize the usage of the cloud's bandwidth, while respecting the QoS constraint. It extends the switching algorithm and takes advantage of the bandwidth estimation formulas to define the cloud's bandwidth allocations at the swarm level.
  To validate the algorithm, we develop two simulators. The first one simulates the default behavior of the seed where each download operation is treated individually and the content is delivered using HTTP. The second simulator simulates the bandwidth distribution and switching algorithm where BT can be used along with HTTP to distribute content. We validate both approaches using a real trace of the Ubuntu One system: We vary the switching constrains and the cloud upload speed limits and measure the degree of improvement in download time of the involved clients using our algorithm (BT and HTTP together) compared to the use of HTTP alone. The results show important improvements in the download time experienced by the peers.

All the aforementioned research contributions have been devised to match the practical requirements of a real-life Personal Cloud system, such as the StackSync prototype.

3 Erasure-Coded Storage with Separate Metadata

Besides well-known benefits, commodity cloud storage also raises concerns that include security, reliability, and economical costs. Erasure coding is a key technique aimed at saving space and retaining robustness against faults in distributed storage systems. In short, an erasure code splits a large data value into n fragments such that from any k of them the input value can be reconstructed. Erasure coding is used by several large-scale storage systems [4, 5] that offer large capacity, high throughput, resilience to faults, and efficient use of storage space.

Whereas the storage systems in production use today only tolerate crashes or outages, storage systems in the Byzantine failure model (BFT) survive also more severe faults, ranging from arbitrary state corruption to malicious attacks on processes. In general, the BFT models untrusted data repositories. Here, we consider a model where multiple clients concurrently access a storage service provided by a distributed set of nodes , where t out of n nodes may be Byzantine. We model the storage service as an abstract read/write register object.

Although BFT erasure-coded distributed storage systems have received some attention in the literature [6, 7, 8, 9, 10], our understanding of their properties is not mature. The role of different quorums, the semantics of concurrent access, the latency of protocols, and the processing capabilities of the nodes have been investigated thoroughly for protocols based on replication [11, 12]; in contrast, our understanding of erasure-coded distributed storage lies far behind. In fact, the existing BFT erasure-coded storage protocols suffer from multiple drawbacks: some require nodes to store an unbounded number of values [6] or rely on node-to-node communication [7], others need computationally expensive public-key cryptography [7, 8] or may block clients due to concurrent operations of other clients [8].

We introduce AWE, the first erasure-coded distributed implementation of a multi-reader multi-writer (MRMW) register that is, at the same time, (1) asynchronous, (2) wait-free, (3) atomic, (4) amnesic, (5) tolerates the optimal number of Byzantine nodes, and (6) does not use public-key cryptography. Although different subsets of these robustness properties have been demonstrated so far, they have never been achieved together for erasure-coded storage, as explained later. Combining these properties, that we describe in the following, has been a longstanding open problem [6].

More specifically, AWE is an asynchronous protocol that provides the strongest liveness and safety properties, namely wait-freedom [13] and atomicity (or linearizability) [14]. Roughly, wait-free liveness means that any correct client operation terminates irrespective of the behavior of the faulty nodes and clients, whereas atomicity means that all operations appear to take effect instantaneously. Moreover, protocol AWE is amnesic [15] in the sense that nodes store a bounded number of values and erase obsolete data.

It has been shown that $n > 3t$ nodes are needed for distributed BFT storage [16], and all known erasure-coded BFT storage protocols actually use $n > 3t$ nodes to store payload (bulk) data. This dramatically increases the cost of BFT over crash-tolerant storage, where less than half of the nodes may be faulty. By distinguishing between metadata (short control information) and bulk data (the erasure-coded stored values) and by introducing two separate classes of nodes that store metadata and bulk data, respectively, AWE beats this bound for the class of data nodes (that store bulk data). In particular, with a k-out-of-n era-

sure code, protocol AWE needs only $2t + k$ data nodes , for any $k \geq 1$. This approach saves resources in practice, as storage costs for the bulk data often dominate, and it resembles the separation between agreement and execution for BFT services [17]. The data nodes may be passive objects that support read and write operations but cannot execute code, as in Disk Paxos [18]. In practice, such services may be provided by the key-value stores (KVS) popular in cloud storage.

We formulate AWE in a modular way using an abstract metadata service that stores control information with an atomic snapshot object. A snapshot object may be realized in a distributed asynchronous system from simple read/write registers [19]. For making this implementation fault-tolerant, these registers must still be emulated from $n > 3t$ different metadata nodes , in order to tolerate t Byzantine nodes.

Finally, AWE uses simple cryptographic hash functions but no expensive public-key operations. To explain the use of cryptography in AWE, we show that separating data from metadata and reducing the number of data nodes to $3t$ or less implies the use cryptographic techniques. This result is interesting in its own right, as it implies that any distributed BFT storage protocol that uses $3t$ or fewer nodes for storing bulk data must involve cryptographic hash functions and place a bound on the computational power of the Byzantine nodes. As all existing BFT erasure-coded storage protocols (including AWE) rely on cryptography, this result does not pose a restriction on practical systems. However, it illustrates a fundamental limitation that is particularly relevant for $k = 1$, i.e., for replication-based BFT storage protocols.

The remaining of this section is composed as follows. The AWE protocol is presented in Section 3.1. The communication and storage complexities of AWE are compared to those of existing protocols in Section 3.2.

3.1 Protocol AWE

This subsection introduces the asynchronous wait-free erasure-coded Byzantine distributed storage protocol (AWE).

3.1.1 Abstractions

Erasure code. An (n, k)-erasure code (EC) with domain $\mathcal{V}$ is given by an encoding algorithm, denoted *Encode*, and a reconstruction algorithm, called *Reconstruct*. We consider only maximum-distance separable codes, which achieve the Singleton bound in the following sense. Given a (large) value $v \in \mathcal{V}$, algorithm $\mathit{Encode}_{k,n}(v)$ produces a vector $[f_1, \ldots, f_n]$ of n fragments, which are from a domain $\mathcal{F}$. A fragment is typically much smaller than the input, and any k fragments contain all information of v, that is, $|\mathcal{V}| \approx k|\mathcal{F}|$. For an n-vector $F \in \left(\mathcal{F} \cup \{\perp\}\right)^n$, whose entries are either fragments or the symbol $\perp$, algorithm $\mathit{Reconstruct}_{k,n}(F)$ outputs a value $v \in \mathcal{V}$ or $\perp$. An output value of $\perp$ means that the reconstruction failed. The completeness property of an erasure code requires that an encoded value can be reconstructed from any k fragments. In other words, for every $v \in \mathcal{V}$, when one computes $F \leftarrow \mathit{Encode}_{k,n}(v)$ and then erases up to $n - k$ entries in F by setting them to $\perp$,

algorithm $Reconstruct_{k,n}(F)$ outputs v. For a more general and high level primer on erasure coding we refer the reader to [20].

Metadata service. The metadata service is implemented by a standard atomic snapshot object [19], called *dir*, that serves as a directory. A snapshot object extends the simple storage function of a register to a service that maintains one value for each client and allows for better coordination. Like an array of multi-reader single-writer (MRSW) registers, it allows every client to update its value individually; for reading it supports a scan operation that returns the vector of the stored values, one for every client. More precisely, the operations of *dir* are:

- An Update operation to *dir* is triggered by an invocation $\langle$ *dir-Update* $\mid c, v \rangle$ by client c that takes a value $v \in \mathcal{V}$ as parameter and terminates by generating a response $\langle$ *r-UpdateAck* $\rangle$ with no parameter.
- A Scan operation on *dir* is triggered by an invocation $\langle$ *dir-Scan* $\rangle$ with no parameter; the snapshot object returns a vector V of $m = |\mathcal{C}|$ values to c as the parameter in the response $\langle$ *r-ScanResp* $\mid V \rangle$, with $V[c] \in \mathcal{V}$ for $c \in \mathcal{C}$.

The sequential specification of the snapshot object follows directly from the specification of an array of m MRSW registers (hence, the snapshot initially stores the special symbol $\perp \notin \mathcal{V}$ in every entry). When accessed concurrently from multiple clients, its operations appear to take place atomically, i.e., they are linearizable. Snapshot objects are weak — they can be implemented from read/write registers [19], which, in turn, can be implemented from a set of a distributed processes subject to Byzantine faults. Wait-free amnesic implementations of registers with the optimal number of $n > 3t$ processes are possible using existing constructions [21, 22].

Data nodes. Data nodes provide a simple key-value store interface. We model the state of data nodes as an array $data[ts] \in \Sigma^*$, initially $\perp$, for $ts \in Timestamps$. Every value is associated to a timestamp, which consists of a sequence number *sn* and the identifier c of the writing client, i.e., $ts = (sn, c) \in Timestamps = N_0 \times (\mathcal{C} \cup \{\perp\})$; timestamps are initialized to $T_0 = (0, \perp)$. Data node d_i exports three operations:

- $\langle$ d_i*-Write* $\mid ts, v \rangle$, which assigns $data[ts] \leftarrow v$ and returns $\langle$ d_i*-WriteAck* $\mid ts \rangle$;
- $\langle$ d_i*-Read* $\mid ts \rangle$, which returns $\langle$ d_i*-ReadResp* $\mid ts, data[ts] \rangle$; and
- $\langle$ d_i*-Free* $\mid TS \rangle$, which assigns $data[ts] \leftarrow \perp$ for all $ts \in TS$, and returns $\langle$ d_i*-FreeAck* $\mid TS \rangle$.

3.1.2 Protocol overview

Protocol AWE uses the metadata directory *dir* to maintain pointers to the fragments stored at the data nodes. As in standard implementations of multi-writer distributed storage [23], every value is associated to a timestamp, which consists of a sequence number *sn* and the identifier c of the writing client, i.e., $ts = (sn, c) \in Timestamps = N_0 \times (\mathcal{C} \cup \{\perp\})$; timestamps

are initialized to $T_0 = (0, \perp)$. The metadata contains the timestamp of the most recently written value for every client, and readers determine the value to read by retrieving all timestamps, determining their maximum, and accessing the fragments associated to the highest timestamp. Comparisons among timestamps use the standard ordering, where $ts_1 > ts_2$ for $ts_1 = (sn_1, c_1)$ and $ts_2 = (sn_2, c_2)$ if and only if $sn_1 > sn_2 \vee (sn_1 = sn_2 \wedge c_1 > c_2)$.

The directory stores an entry for every writer; it contains the timestamp of its most recently written value, the identities of those nodes that have acknowledged to store a fragment of it, a vector with the hashes of the fragments for ensuring data integrity, and additional metadata to support concurrent reads and writes. The linearizable semantics of protocol AWE are obtained from the atomicity of the metadata directory.

At a high level, the writer first invokes *dir-Scan* on the metadata to read the highest stored timestamp, increments it, and uses this as the timestamp of the value to be written. Then it encodes the value to n fragments and sends one fragment to each data node. The data nodes store it and acknowledge the write. After the writer has received acknowledgments from $t + k$ data nodes, it writes their identities (together with the timestamp and the hashes of the fragments) to the metadata through *dir-Update*. The reader proceeds accordingly: it first invokes *dir-Scan* to obtain the entries of all writers; it determines the highest timestamp among them and extracts the fragment hashes and the identities of the data nodes; finally, it contacts the data nodes and reconstructs the value after obtaining k fragments that match the hashes in the metadata.

Although this simple algorithm achieves atomic semantics, it does not address timely garbage-collection of obsolete fragments, the main problem to be solved for amnesic erasure-code distributed storage. If a writer would simply replace the fragments with those of the value written next, it is easy to see that a concurrent reader may stall.

Protocol AWE uses two mechanisms to address this: first, the writer retains those values that may be accessed concurrently and exempts them from garbage collection so that their fragments remain intact for concurrent readers, which gives the reader enough time to retrieve its fragments. Secondly, some of the retained values may also be frozen in response to concurrent reads; this forces a concurrent read to retrieve a value that is guaranteed to exist at the data nodes rather than simply the newest value, thereby effectively limiting the amount of stored values. A similar freezing method has been used for wait-free atomic storage with replicated data [21, 22], but it must be changed for erasure-coded storage with separated metadata. The retention technique together with the separation of metadata appears novel. More specifically, metadata separation prevents straightforward applications of existing "freezing" techniques, whereas storage that is simultaneously wait-free and amnesic requires garbage collection method that we devised as part of the research of CloudSpaces.

For the two mechanisms, i.e., retention and freezing, every reader maintains a reader index, both in its local variable *readindex* and in its metadata. The reader index serves for coordination between the reader and the writers. The reader increments its index whenever it starts a new *r-Read* and immediately writes it to *dir*, thereby announcing its intent to read. Writers access the reader indices after updating the metadata for a write and before (potentially) erasing obsolete fragments. Every writer w maintains a table *frozenindex* with its most recent recollection of all reader indices. When the newly obtained index of a reader c has changed, then w detects that c has started a new operation at some time after the last write of w.

When w detects a new operation of c, it does not know whether c has retrieved the timestamp from *dir* before or after the *dir-Update* of the current write. The reader may access either value; the writer therefore retains both the current and the preceding value for c by storing a pointer to them in *frozenptrlist* and in *reservedptrlist*. Clearly, both values have to be excluded from garbage collection by w in order to guarantee that the reader completes.

However, the operation of the reader c may access *dir* after the *dir-Update* of one or more subsequent write operation by w, which means that the nodes would have to retain every value subsequently written by w as well. To prevent this from happening and to limit the number of stored values, w freezes the currently written timestamp (as well as the value) and forces c to read this timestamp when it accesses *dir* within the same operation. In particular, the writer stores the current timestamp in *frozenptrlist* at index c and updates the reader index of c in *frozenindex*; then, the writer pushes both tables, *frozenindex* and *frozenptrlist*, to the metadata service during its next *r-Write*. The values designated by *frozenptrlist* (they are called frozen) and *reservedptrlist* (they are called reserved) are retained and excluded from garbage collection until w detects the next read of c, i.e., the reader index of c increases. Thus, the current read may span many concurrent writes of w and the fragments remain available until c finishes reading.

On the other hand, a reader must consider frozen values. When a slow read operation spans multiple concurrent writes, the reader c learns that it should retrieve the frozen value through its entry in the *frozenindex* table of the writer.

The protocol is amnesic because each writer retains at most two values per reader, a frozen value and a reserved value. Every data node therefore stores at most two fragments for every reader-writer pair plus the fragment from the currently written value. The combination of freezing and retentions ensures wait-freedom.

Protocol details are available in the Technical Report [24].

Remarks. AWE does not rely on a majority of correct data nodes for correctness, as this is encapsulated in the directory service. For liveness, though, the protocol needs responses from $t + k$ data nodes during write operations, which is only possible if $n \geq 2t + k$. Furthermore, several optimizations may reduce the storage overhead in practice, e.g., readers can clean up values that are no longer needed by anyone.

3.2 Complexity comparison

This section compares the communication and storage complexities of AWE to existing erasure-coded distributed storage solutions, in a setting with n data nodes and m clients. We denote the size of each stored value $v \in \mathcal{V}$ by $\ell = \lceil \log_2 |\mathcal{V}| \rceil$. In line with the intended deployment scenarios, we assume that ℓ is much larger (by several orders of magnitude) than n^2 and m^2 , i.e., $\ell \gg n^2$ and $\ell \gg m^2$.

We examine the worst-case communication and storage costs incurred by a client in protocol AWE and distinguish metadata operations (on *dir*) from operations on the data nodes. The metadata of one value written to *dir* consists of a pointer, containing the cross checksum with n hash values, the $t + k$ identities of the data nodes that store a data fragment, and a

Table 3.2a Comparison of the communication and space complexities of erasure-coded distributed storage solutions. There are m clients, n data nodes, the erasure code parameter is $k = n - 2t$, and the data values are of size ℓ bits. An asterisk (*) denotes optimal properties.

Protocol	**Communication cost**		**Storage cost**
	Write	*Read*	
ORCAS-A [9]	$(1+m)n\ell$	$2n\ell$	$n\ell$
ORCAS-B [9]	$(1+m)n\ell/k$	$2n\ell/k$	$mn\ell/k$
CASGC [25]	$n\ell/k$ *	∞	$mn\ell/k$
CT [7]	$(n+m)n\ell/(k+t)$	ℓ *	$n\ell/(k+t)$ *
HGR [8]	$n\ell/k$ *	∞	$mn\ell/k$
M-PoWerStore [26]	$n\ell/k$ *	$n\ell/k$	∞
DepSky [10]	$n\ell/k$ *	$n\ell/k$	∞
AWE (Sec. 4.3)	$n\ell/k$ *	$(t+k)\ell/k$	$2m^2n\ell/k$

timestamp. Moreover, the metadata entry of one writer contains also the list of m pointers to frozen values, the m indices relating to the frozen values, and the writer's reader index. Assuming a collision-resistant hash function with output size λ bits and timestamps no larger than λ bits, the total size of the metadata is $O(m^2n\lambda)$.

In the remainder of this section, the size of the metadata is considered to be negligible and is ignored, though it would incur in practice.

According to the above assumption, the complexity of AWE is dominated by the data itself. When writing a value $v \in \mathcal{V}$, the writer sends a fragment of size ℓ/k and a timestamp of size λ to each of the n data nodes. Assuming further that $\ell \gg \lambda$, the total storage space occupied by v at the data nodes amounts to $n\ell/k$ bits. Similarly, a read operation incurs a communication cost of $(t+k)k/\ell$ bits. With respect to storage complexity, protocol AWE freezes and reserves two timestamps and their fragments for each writer-reader pair, and additionally stores the fragments of the last written value for each writer. This means that the storage cost is at most $2m^2n\ell/k$ bits in total.

Table 3.2a shows the communication and storage costs of protocol AWE and the related protocols. Observe that in CASGC [25] and HGR [8], a read operation concurrent with an unbounded number of writes may not terminate, hence we state their cost as ∞. Moreover, in contrast to AWE, DepSky [10] is neither wait-free nor amnesic and M-PoWerStore [26] is not amnesic. It is easy to see that the communication complexity of AWE is lower than that of most storage solutions.

4 Hybris: Efficient and Robust Hybrid Cloud Storage

4.1 Introduction

Hybrid cloud storage entails storing data on private premises as well as on one (or more) remote, public cloud storage providers. To enterprises, such hybrid design brings the best of both worlds: the benefits of public cloud storage (e.g., elasticity, flexible payment schemes and disaster-safe durability) as well as the control over enterprise data. For example, an enterprise can keep the sensitive data on premises while storing less sensitive data at potentially untrusted public clouds. In a sense, hybrid cloud eliminates to a large extent various security concerns that companies have with entrusting their data to commercial clouds [3] — as a result, enterprise-class hybrid cloud storage solutions are booming with all leading storage providers, such as EMC[4], IBM[5], Microsoft[6] and others, offering their proprietary solutions.

That said, cloud storage concerns do not end with security and trust. Other potential issues with commodity cloud storage are related to provider reliability, availability and performance, vendor lock-in concerns, as well as consistency, as cloud storage services are notorious for providing only eventual consistency [27]. To this end, several research works (e.g., [28, 29, 30, 31]) considered storing data robustly into public clouds, by leveraging multiple commodity cloud providers. In short, the idea behind these multi-cloud storage systems such as DepSky [28], ICStore [29] and SPANStore [30] and SCFS [31] is to leverage multiple public cloud providers with the goals of distributing the trust across clouds, increasing reliability, availability and consistency guarantees, and/or optimizing the cost of using the cloud. A significant advantage of the multi-cloud 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, the existing robust multi-cloud storage systems suffer from serious limitations. Often, the robustness of these systems is limited to tolerating cloud outages, but not arbitrary or malicious behavior in clouds (e.g., data corruptions) — this is the case with ICStore [29] and SPANStore [30]. Other multi-cloud systems that do address malice in systems (e.g., DepSky [28] and SCFS [31]) require prohibitive cost of relying on $3f + 1$ clouds to mask f faulty ones. This is a major overhead with respect to tolerating only cloud outages, which makes these systems expensive to use in practice. Moreover, all existing multi-cloud storage systems scatter storage metadata across public clouds increasing the difficulty of storage management and impacting performance.

We unify the hybrid cloud approach with that of robust multi-cloud storage and present Hybris, the first robust hybrid cloud storage system. Hybris effectively brings together the best of both worlds, increasing security, reliability and consistency. At the same time, the novel design of Hybris allows for the first time to tolerate potentially malicious clouds at the price of tolerating only cloud outages.

Hybris exposes the de-facto standard key-value store API and is designed to seamlessly

[3] See e.g., `http://vmw.re/1Ja4fvI`.

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

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

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

replace services such as Amazon S3 as the storage back-end of modern cloud applications. The key idea behind Hybris is that it keeps all storage metadata on private premises, even when those metadata pertain to data outsourced to public clouds. This approach not only allows more control over the data scattered around different public clouds, but also allows Hybris to significantly outperform existing robust public multi-cloud storage systems, both in terms of system performance (e.g., latency) and storage cost, while providing strong consistency guarantees. The salient features of Hybris are as follows:

- Tolerating untrusted clouds at the price of outages. Hybris puts no trust in any given public cloud provider; namely, Hybris can mask arbitrary (including malicious) faults of up to f public clouds. Interestingly, Hybris replicates data on as few as $f+1$ clouds in the common case (when the system is synchronous and without faults), using up to f additional clouds in the worst case (e.g., network partitions, cloud inconsistencies and faults). This is in sharp contrast to existing multi-cloud storage systems that involve up to $3f+1$ clouds to mask f malicious ones (e.g., [28, 31]).
- Efficiency. Hybris is efficient and incurs low cost. In common case, a Hybris write involves as few as $f+1$ public clouds, whereas reads involve only a single cloud, despite the fact that clouds are untrusted. Hybris achieves this without relying on expensive cryptographic primitives; indeed, in masking malicious faults, Hybris relies solely on cryptographic hashes. Besides, by storing metadata locally on private cloud premises, Hybris avoids expensive round-trips for metadata operations that plagued previous multi-cloud storage systems. Finally, to reduce replication overhead, Hybris optionally supports erasure coding, along the guidelines developed with protocol AWE (Section 3).
- Scalability. The potential pitfall of using private cloud in combination with public clouds is in incurring a scalability bottleneck at a private cloud. Hybris avoids this pitfall by keeping the metadata very small. As an illustration, the replicated variant of Hybris maintains about 50 bytes of metadata per key, which is an order of magnitude less than comparable systems [28]. As a result, Hybris metadata service residing on a small commodity private cloud, can easily support up to 30k write ops/s and nearly 200k read ops/s, despite being fully replicated for metadata fault-tolerance.

Indeed, for Hybris to be truly robust, it has to replicate metadata reliably. Given inherent trust in private premises, we assume faults within private premises that can affect Hybris metadata to be crash-only. To maintain the Hybris footprint small and to facilitate its adoption, we chose to replicate Hybris metadata layering Hybris on top of Apache ZooKeeper coordination service [32]. Hybris clients act simply as ZooKeeper clients — our system does not entail any modifications to ZooKeeper, hence facilitating Hybris deployment. In addition, we designed Hybris metadata service to be easily portable to SQL-based replicated RDBMS as well as NoSQL data stores that export conditional update operation (e.g., HBase or MongoDB), which can then serve as alternatives to ZooKeeper.

Hybris offers full fledged per-key multi-writer multi-reader capabilities that guarantees linearizability (atomicity) [33] of reads and writes even in presence of eventually consistent public clouds [27]. To achieve this, Hybris relies on strong metadata consistency within a private cloud to mask potential inconsistencies at public clouds — in fact, Hybris treats cloud inconsistencies simply as arbitrary cloud fault. Furthermore, our implementation of the

Hybris metadata service over Apache Zookeeper is interesting in its own right as it uses lock-free (wait-free [34]) concurrency control that further boosts the scalability of our system with respect to lock-based systems such as SPANStore [30], DepSky [28] and SCFS [31].

Finally, Hybris optionally supports the following optional features:

- Caching of data stored at public clouds. While different caching solutions can be applied to Hybris, we chose to interface Hybris with Memcached[7] distributed cache, with Memcached deployed on the same machines that run ZooKeeper servers;
- Symmetric-key encryption for data confidentiality, leveraging trusted Hybris metadata to store and share cryptographic keys;
- Tunable consistency. In alternative to strong consistency, Hybris can offer improved performance by relaxing in a controlled and adjustable manner the consistency semantics.

We implemented Hybris in Java[8] and evaluated it using both microbenchmarks and the YCSB [35] macrobenchmark. Our evaluation shows that Hybris significantly outperforms state-of-the-art robust multi-cloud storage systems, with a fraction of the cost and stronger consistency.

The rest of the section is organized as follows. In Section 4.2, we present the Hybris architecture and system model. Then, in Section 4.3, we give the algorithmic aspects of the Hybris protocol. In Section 4.4 we discuss Hybris implementation and optimizations.

4.2 Hybris overview

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

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

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

Finally, at the heart of the system is the Hybris client, whose library is responsible for interactions with public clouds, RMDS and the caching service. Hybris clients are also re-

[7] `http://memcached.org/`.
[8] Hybris code is available at `https://github.com/pviotti/hybris`.

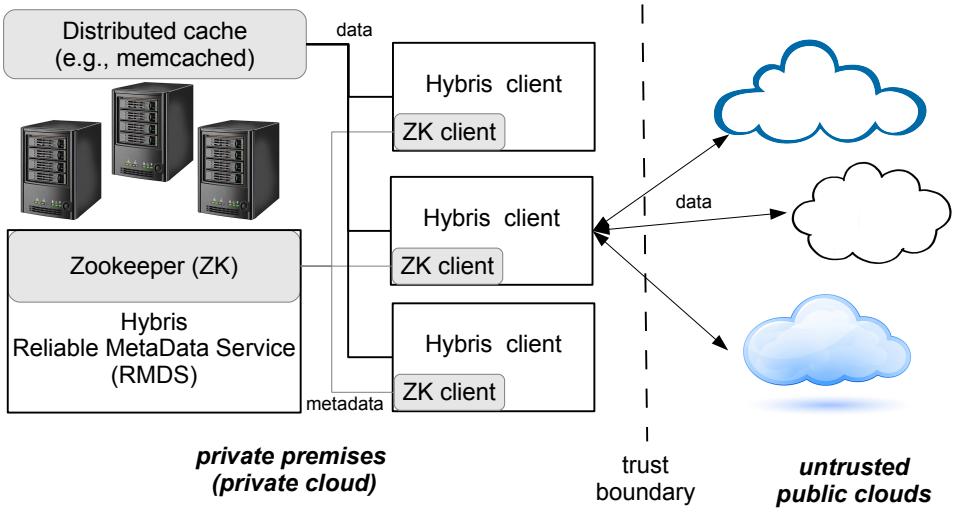


Figure 1: Hybris architecture. Reused (open-source) components are depicted in grey.

sponsible for encrypting and decrypting data in case data confidentiality is enabled — in this case, clients leverage RMDS for sharing encryption keys (see Sec. 4.3.8).

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

System model. We assume an unreliable distributed system where any of the components might fail. In particular, we consider dual fault model, where: (i) the processes on private premises (i.e., in the private cloud) can fail by crashing, and (ii) we model public clouds as potentially malicious (i.e., arbitrary-fault prone [36]) processes. Processes that do not fail are called *correct*.

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

Similarly to our fault model, our communication model is dual, with the model boundary coinciding with our trust boundary (see Fig. 1).[9] Namely, we assume that the communication among processes located in the private portion of the cloud is partially synchronous [37] (i.e., with arbitrary but finite periods of asynchrony), whereas the communication among clients and public clouds is entirely asynchronous (i.e., does not rely on any timing assumption) yet reliable, with messages between correct clients and clouds being eventually delivered.

Our consistency model is likewise dual. We model processes on private premises as classical state machines, with their computation proceeding in indivisible, atomic steps. On the other hand, we model clouds as eventually consistent [27]; 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.

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

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 the cryptographic hash functions we use (SHA-1), and that it cannot spoof the communication among non-malicious processes.

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

Hybris implements a multi-writer multi-reader key-value storage. Hybris is strongly consistent, i.e., it implements atomic (or linearizable [33]) semantics. In distributed storage context, atomicity provides an illusion that a complete operation op is executed instantly at some point in time between its invocation and response, whereas the operations invoked by faulty clients appear either as complete or not invoked at all.

Despite providing strong consistency, Hybris is highly available. Hybris writes by a correct client are guaranteed to eventually complete [34]. On the other hand, Hybris guarantees a read operation by a correct client to complete always, except in an obscure corner case where there is an infinite number of writes to the same key concurrent with the read operation.

4.3 Hybris Protocol

4.3.1 Overview

The key component of Hybris is RMDS which maintains metadata associated with each key-value pair. In the vein of Farsite [38], Hybris RMDS maintains pointers to data locations and cryptographic hashes of the data. However, unlike Farsite, RMDS additionally includes a client-managed logical timestamp for concurrency control, as well as data size.

Such Hybris metadata, despite being lightweight, is powerful enough to enable tolerating arbitrary cloud failures. Intuitively, the cryptographic hash within a trusted and consistent RMDS enables end-to-end integrity protection: it ensures that neither corrupted values produced by malicious clouds, nor stale values retrieved from inconsistent clouds, are ever returned to the application. Complementarily, data size helps prevent certain denial-of-service attack vectors by a malicious cloud (see Sec. 4.4.2).

Furthermore, Hybris metadata acts as a directory pointing to $f + 1$ clouds that have been previously updated, enabling a client to retrieve the correct value despite f of them being arbitrary faulty. In fact, with Hybris, as few as $f + 1$ clouds are sufficient to ensure both consistency and availability of read operations (namely GET) — indeed, Hybris GET never involves more than $f + 1$ clouds (see Sec. 4.3.3). Additional f clouds (totaling $2f + 1$ clouds) are only needed to guarantee that write operations (namely PUT) are available as well (see Sec. 4.3.2). Note that since f clouds can be faulty, and a value needs to be stored in $f + 1$

clouds for durability, overall $2f + 1$ clouds are required for PUT operations to be available in the presence of f cloud outages.

Finally, besides cryptographic hash and pointers to clouds, metadata includes a timestamp that, roughly speaking, induces a partial order of operations which captures the real-time precedence ordering among operations (atomic consistency). The subtlety of Hybris (see Sec. 4.3.6 for details) is in the way it combines timestamp-based lock-free multi-writer concurrency control within RMDS with garbage collection (Sec. 4.3.5) of stale values from public clouds to save on storage costs.

In the following we detail each Hybris operation individually.

4.3.2 PUT Protocol

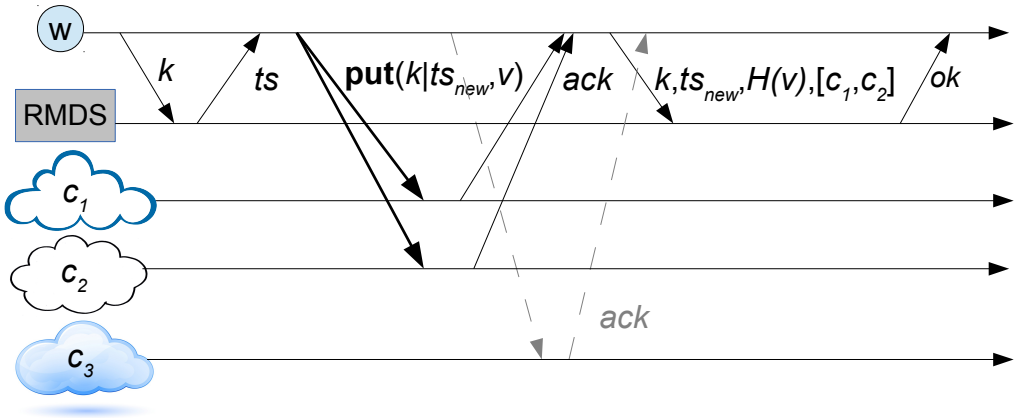


Figure 2: Hybris PUT protocol illustration ($f = 1$). Common-case communication is depicted in solid lines.

Hybris PUT protocol entails a sequence of consecutive steps illustrated in Figure 2. To write a value v under key k, a client first fetches from RMDS the latest authoritative timestamp ts by requesting the metadata associated with key k. Timestamp ts is a tuple consisting of a sequence number sn and a client id cid. Based on timestamp ts, the client computes a new timestamp ts_{new}, whose value is $(sn + 1, cid)$. Next, the client combines the key k and timestamp ts_{new} to a new key $k_{new} = k|ts_{new}$ and invokes **put** (k_{new}, v) on $f + 1$ clouds in parallel. Concurrently, the clients start 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 since this final step is the linearization point of PUT it has to be performed in a specific way as discussed below.

Namely, if the client performs a straightforward update of metadata in RMDS, then it may occur that stored metadata is overwritten by metadata with a lower timestamp (old-new inversion), breaking the timestamp ordering of operations and Hybris consistency. To solve the old-new inversion problem, we require RMDS to export an atomic conditional update operation. Then, in the final step of Hybris PUT, the client issues conditional update to RMDS which updates the metadata for key k only if the written timestamp ts_{new} is greater than the timestamp for key k that RMDS already stores. In Section 4.4 we describe how we implement this functionality over Apache ZooKeeper API; alternatively other NoSQL and SQL DBMSs that support conditional updates can be used.

4.3.3 GET in the common case

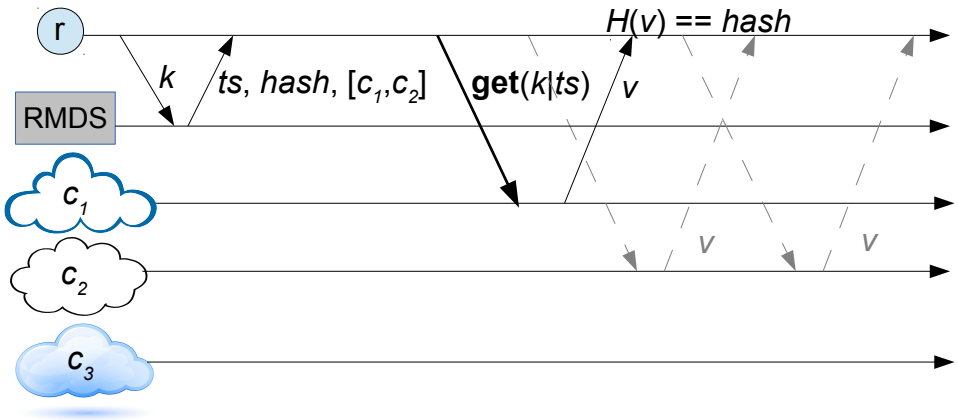


Figure 3: Hybris GET protocol illustration ($f = 1$). Common-case communication is depicted in solid lines.

Hybris GET protocol is illustrated in Figure 3. 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 download the data from the second cloud from $cloudList$ (see dashed arrows in Fig. 3) and so on, until the client exhausts all $f + 1$ clouds from $cloudList$.[10]

In specific corner cases, caused by concurrent garbage collection (described in Sec. 4.3.5),

[10] As we discuss in details in Section 4.4, in our implementation, clouds in $cloudList$ are ranked by the client by their typical latency in the ascending order, i.e., when reading the client will first read from the "fastest" cloud from $cloudList$ and then proceed to slower clouds.

failures, repeated timeouts (asynchrony), or clouds' inconsistency, the client has to take additional actions in GET (described in Sec. 4.3.6).

4.3.4 Transactional PUT

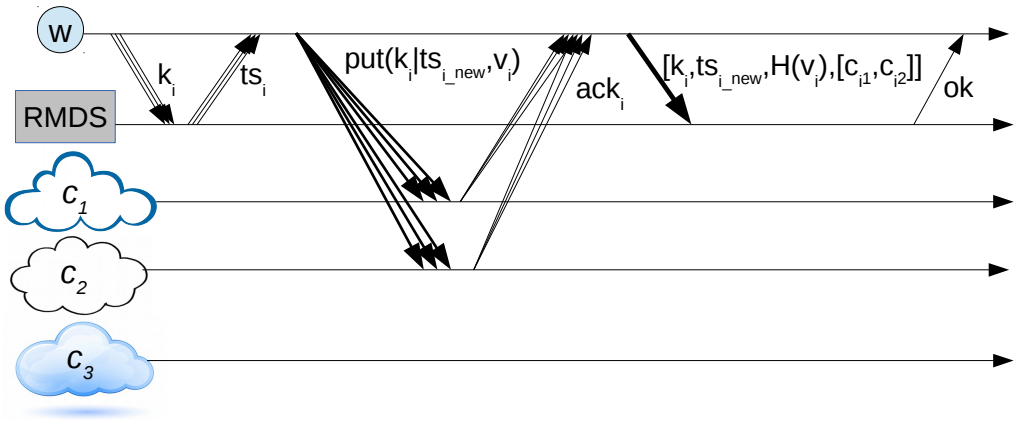


Figure 4: Hybris transactional PUT protocol illustration ($f = 1$). Worst case communication patterns are omitted for clarity.

The transactional PUT operation is exposed by the Hybris library in order to enable atomic writing to different keys. The sequence of consecutive steps associated with the transactional PUT operation is depicted in Figure 4. Similarly to the normal PUT protocol, at first the client fetches the latest authoritative timestamps ($[ts_0...ts_n]$) by requesting in parallel to the RMDS the metadata associated to the keys it wants to write ($[k_0...k_n]$). 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 whose expirations are set according to the typically upload latencies observed for the given payload sizes. In the common case, the $f + 1$ clouds reply to the the client for each key in a timely fashion, before the timer expires. Otherwise, the client invokes **put** (k_{i_new}, v_i) on up to f secondary clouds (this worst case scenario is not shown in Fig. 4 for clarity). Once the client has received acks from $f + 1$ different clouds for each key, it is assured that the transactional PUT is durable and it can thus proceed to the final stage of the operation.

In the final step, the client attempts to store in RMDS the updated metadata associated with each key written k_i, consisting of the timestamp ts_{i_new}, the cryptographic hash $H(v_i)$, size of value v_i $size(v_i)$, and the list of pointers to those $f + 1$ clouds that have acknowledged storage of value v_i. As for the normal PUT operation, in order to make the whole operation linearizable thus avoiding the so-called old-new inversion anomaly, we employ the conditional update exposed by the RMDS: the metadata update for each key k_i succeeds only if the written timestamp ts_{i_new} is greater than the timestamp for key k_i that RMDS already stores. Besides, in order to enforce the transactional atomicity of the set of write operations, we wrap the metadata updates into an RMDS transaction. Specifically, we use the MULTI API exposed by Apache ZooKeeper, which implements the requested funcionality. Hence,

if any of the single write to RMDS fails, the whole transaction fails, obliterating the entire Hybris transactional PUT. In case of failure, the value possibly written on the cloud stores are then erased by the periodical garbage collection background task.

4.3.5 Garbage Collection

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

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

4.3.6 GET in the worst-case

In the context of cloud storage, there are known issues with weak, e.g., eventual [27] 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 4.3.3 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 an 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. 4.3.5).

To cope with these issues and eventual consistency in particular, Hybris leverages metadata service consistency to mask data inconsistencies in the clouds effectively allowing availability to be traded off for consistency. To this end, Hybris client indulgently reiterates the GET by reissuing a **get** to all clouds in parallel, and waiting to receive at least one value matching the desired hash. However, due to possible concurrent garbage collection (Sec. 4.3.5), a client needs to make sure it always compares the values received from clouds to the most recent key metadata. This can be achieved in two ways: (i) by simply looping the entire GET including metadata retrieval from RMDS, or (ii) by looping only **get** operations at $f + 1$ clouds while fetching metadata from RMDS only when metadata actually changes.

In Hybris, we use the second approach. Notice that this suggests that RMDS must be able to inform the client proactively about metadata changes. This can be achieved by having a RMDS that supports subscriptions to metadata updates, which is possible to achieve in, e.g.., Apache ZooKeeper (using the concepts of watches, see Sec. 4.4 for details). The entire protocol executed only if common-case GET fails (Sec. 4.3.3) 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[11].
   (a) If the hash verification succeeds, the GET returns v_c.
   (b) Otherwise, the client discards v_c and reissues **get** $(k|ts)$ at cloud c.
4. At any point in time, if the client receives a metadata update notification for key k from RMDS, the client cancels all pending downloads, and repeats the procedure by going to step 1.

The complete Hybris GET, as described above, ensures finite-write termination [39] in presence of eventually consistent clouds. Namely, a GET may fail to return a value only theoretically, in case of infinite number of concurrent writes to the same key, in which case the garbage collection at clouds (Sec. 4.3.5) might systematically and indefinitely often remove the written values before the client manages to retrieve them.[12]

4.3.7 DELETE and LIST

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

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

4.3.8 Confidentiality

Adding confidentiality to Hybris is straightforward.To this end, during a PUT, just before uploading data to $f+1$ public clouds, the client encrypts the data with a symmetric cryptographic key k_{enc}. Then, in the final step of the PUT protocol (see Sec. 4.3.2), when the client writes metadata to RMDS using conditional update, the client simply adds k_{enc} to metadata and computes the hash on ciphertext (rather than on cleartext). The rest of the PUT protocol remains unchanged. The client may generate a new key with each new encryption, or fetch the last used key from the metadata service, at the same time it fetches the last used timestamp.

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

[11] For simplicity, we model the absence of a value as a special NULL value that can be hashed.
[12] Notice that it is straightforward to modify Hybris to guarantee read availability even in case of an infinite number of concurrent writes, by switching off the garbage collection.

4.3.9 Erasure coding

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

Deriving an erasure coding variant of Hybris follows the scheme of protocol AWE, developed in Section 3. Namely, in a PUT, the client encodes original data into $f + k$ erasure coded chunks and places one chunk 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 one in case of replication) that are stored in the RMDS as the part of metadata. Finally, erasure coded GET involves fetching chunks from k clouds in common case, with chunk hashes verified against those stored in RMDS. In the worst case, Hybris with erasure coding uses up to $2f + k$ (resp., $f + k$) clouds in PUT (resp., GET).

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

4.3.10 Tunable consistency

As the rise of the NoSQL movement testified, not every application can afford the performance drawbacks entailed by strong consistency. Besides, a maybe surprising large amount of nowadays applications actually requires weaker correctness semantics. For instance, recent work [40] shown that most applications directly facing end users are well served by causally consistent storage systems [41]. Hence, a wide and multi-dimensional spectrum of different consistency semantics has been developed over the years both by the database and the distributed systems research communities.

In order to better serve this diversified range of needs, Hybris implements and exposes tunable consistency semantics. Namely, for each execution it is possible to make Hybris respect two consistency models alternative to linearizability, i.e., read-your-write consistency and bounded staleness. Formally, read-your-write consistency [42] mandates that a read operation invoked by a process can be applied only on replicas of the storage system that have already performed all write operations previously issued by the same process. Bounded staleness, as the name hints, is a condition that restricts the staleness of the information read from a storage system. Several different models based on the concept of bounded staleness have been proposed in literature. Some of them measure the staleness in terms of data versions [43], whereas some others express it as function of the real-time passed since the corresponding write operation [44].

Read-your-write is a very weak condition that only requires the reader to obtain results that include its own writes. This condition is easily implemented in Hybris by leveraging caching. Essentially, an on-write caching policy is enabled in order to cache all the data written by each client. In particular, upon a successful write operation, a client stores the written data in Memcached as associated to the key used to write data to the cloud stores (e.g., $\langle k|ts_{new}\rangle$), which identifies its unique write operation. Additionally, the client caches

in its own local memory the aforementioned key associated to the original key (i.e., k). Successive reads will first try to fetch the data from the caching layer by using the operation key cached locally, thus possibly obtaining values previously written without incurring into the monetary and latency costs of communications with cloud stores.

Similarly, for bounded staleness we use the caching layer composed by Memcached servers. In particular, in order to implement the time based bounded staleness restriction we set the expiration date of each cached value to a certain predefined Δ. Besides, since according to this policy clients can read each other's writes, we write the cached value on Memcached under the original key k instead of using, as in the read-your-write semantic, the key identifying the single write operation.

Version based bounded staleness is implemented in Hybris by using caching and by adding to the metadata stored on RMDS for each key a field that accounts for the number of versions written from the last caching operation. Upon writing, the client reads from RMDS the number of version elapsed since the last caching operation for key being written. In case of successful write, if the number of non-cached versions exceeds a predefined threshold η, the value is cached on Memcached under its original key (i.e., k). 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.

Although in principle Hybris architecture does not prevent the implementation of other consistency models, some of them require further computational capabilities or native handling of causal semantics from the RMDS component. Hence, in order to stand by our choice of keeping Hybris codebase simple and easy to use, we preferred to only implement the two aforementioned semantics as they are good representative of popular tradeoffs that favor performance requirements over consistency.

Besides, as future work, we envison the implementation of different consistency policies on a per-operation basis. Thus, much like systems supporting RedBlue consistency [45], Hybris would offer the possibility of performing operations supporting strong consistency rather than other more relaxed semantics.

4.4 Implementation

We implemented Hybris in Java. The implementation pertains solely to the Hybris client side since the entire functionality of the metadata service (RMDS) is layered on top of Apache ZooKeeper client. Namely, Hybris does not entail any modification to the ZooKeeper server side. Our Hybris client is lightweight and consists of about 3400 lines of Java code. Hybris client interactions with public clouds are implemented by wrapping individual native Java SDK clients (drivers) for each particular cloud storage provider[13] into a common lightweight interface that masks the small differences across native client libraries.

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

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

4.4.1 ZooKeeper-based RMDS

We layered Hybris implementation over Apache ZooKeeper [32]. In particular, we durably store Hybris metadata as ZooKeeper znodes; in ZooKeeper znodes are data objects addressed by paths in a hierarchical namespace. In particular, for each instance of Hybris, we generate a root znode. Then, the metadata pertaining to Hybris container *cont* is stored under ZooKeeper path $\langle root \rangle / cont$. In principle, for each Hybris key k in container *cont*, we store a znode with path $path_k = \langle root \rangle / cont / k$.

ZooKeeper exports a fairly modest API to its applications. The ZooKeeper API calls relevant to us here are: (i) **create/setData**($p, data$), which creates/updates znode with path p containing *data*, (ii) **getData**(p) to retrieve data stores under znode with p, and (iii) **sync**(), which synchronizes a ZooKeeper replica that maintains the client's session with ZooKeeper leader. Only reads that follow after **sync**() will be atomic.[14]

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

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

In the final step of PUT, the client issues **setData**($path_k, md, vn$) which succeeds only if the znode $path_k$ version is still *vn*. If the ZooKeeper version of $path_k$ changed, the client retrieves the new authoritative Hybris timestamp ts_{last} and compares it to *ts*. If $ts_{last} > ts$, the client simply completes a PUT (which appears as immediately overwritten by a later PUT with ts_{last}). In case, $ts_{last} < ts$, the client retries the last step of PUT with ZooKeeper version number vn_{last} that corresponds to ts_{last}. This scheme (that we believe to be interesting in its own right) is wait-free [34] and is guaranteed to terminate since only a finite number of concurrent PUT operations use a timestamp smaller than *ts*.

Hybris GET. In interacting with RMDS during GET, Hybris client simply needs to make sure its metadata is read atomically. To this end, a client always issues a **sync**() followed by **getData**($path_k$), just like in our PUT protocol. In addition, for subscriptions for metadata updates in GET (Sec. 4.3.6) we use the concept of ZooKeeper watches (set by e.g., **getData**) which are subscriptions on znode update notifications. We use these notifications in Step 4 of the algorithm described in Section 4.3.6.

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

4.4.2 Optimizations

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

Erasure coding. Hybris integrates an optimally efficient Reed-Solomon codes implementation, using the Jerasure library [46], by means of its JNI bindings. The cloud latency rank optimizations remains in place with erasure coding. When performing a PUT, $f + k$ erasure coded blocks are stores in $f + k$ clouds with lowest latency, whereas with GET, $k > 1$ clouds with lowest latency are selected (out of $f + k$ clouds storing data chunks).

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

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

4.4.3 Hybris source code

Hybris Java code base is composed by two main modules: MdsManager and KvsManager, the first is a thin wrapper layer of the metadata distributed storage service (i.e. ZooKeeper), while the latter implements the storage primitives towards the APIs of the main public cloud storage services - currently, it supports Amazon S3, Google Cloud Storage, Rackspace Cloud Files and Windows Azure Blob. Maven is used to compile the code and managing the dependencies.

Hybris source code is publicly released under the terms of Apache 2.0 license and available for download at this web address: `https://github.com/pviotti/hybris`. Erasure coding support is provided by the Jerasure library[15] through its JNI bindings[16].

A thorough evaluation of Hybris can be found in our 2014 Symposium on Cloud Computing (SoCC) paper [47].

4.5 Hybris as StackSync storage backend

StackSync is a Personal Cloud and data synchronization solution developed by URV for the CloudSpaces project. It consists of a full fledged and scalable framework of open source components that operates in a coordinated fashion in order to offer a synchronization service for mobile, web and desktop platforms that, thanks to its architecture, can easily scale. For further details about StackSync we refer the reader to D2.2 or to the paper published in the proceedings of 2014 Middleware Conference [48].

In order to finalize the integration of Hybris with the StackSync prototype, Eurecom has developed a plugin that allows StackSync clients to store data on commercial cloud key-value stores through Hybris. An high level overview of the resulting architecture is depicted in Fig. 5. This integration brings about the benefits of both worlds: an agile and scalable framework for data synchronization, complemented by a failure tolerant and secure cloud-based storage system.

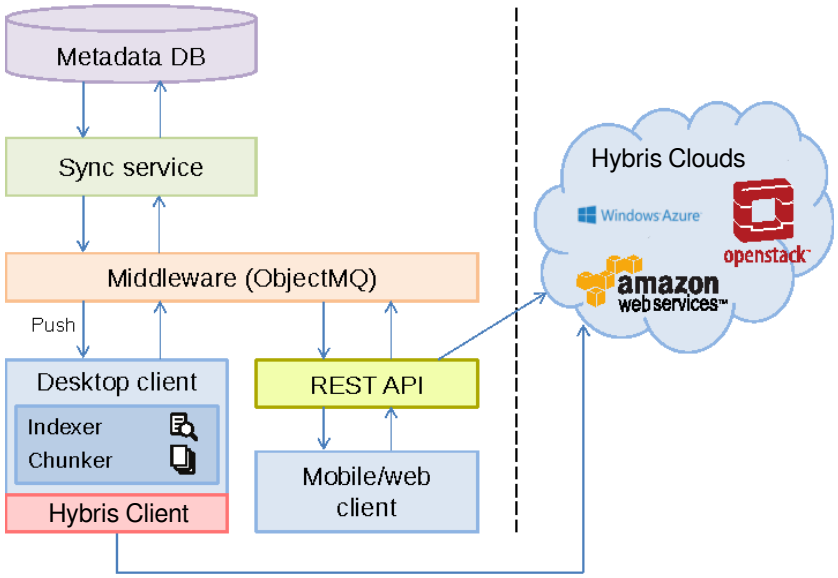


Figure 5: StackSync architecture overview with Hybris.

Beside the plugin, a graphical configuration panel has as well been developed. As illustrated in Fig. 6, such panel allows users to set and tune all aspects of Hybris configuration, such as, for instance, the ZooKeeper address, the caching and the erasure coding optional features.

[15] `http://web.eecs.utk.edu/%7Eplank/plank/papers/CS-08-627.html`
[16] `https://github.com/jvandertil/Jerasure`

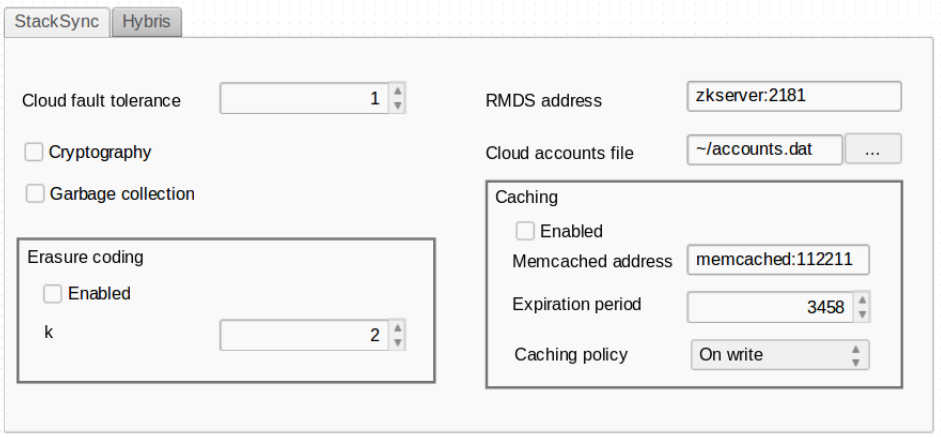


Figure 6: Hybris GUI configuration panel for StackSync.

The source code of StackSync featuring Hybris as storage backend is hosted at this address: `https://github.com/pviotti/stacksync-desktop`.

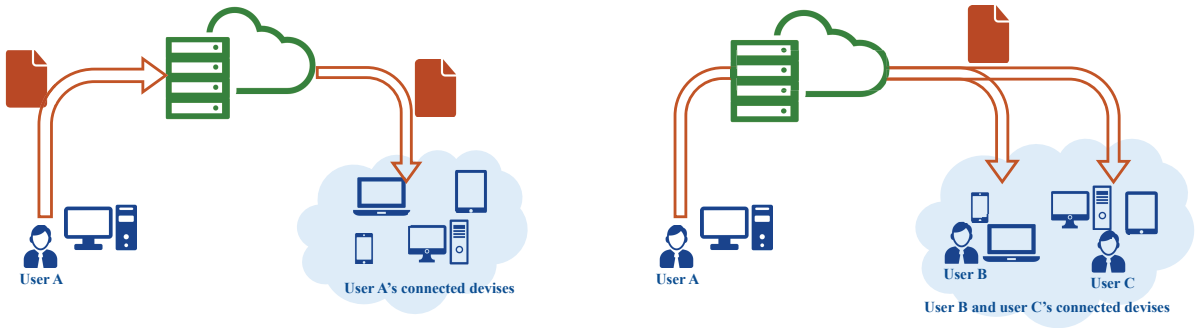
5 BitTorrent in Personal Clouds

Nowadays, users are unceasingly relying on cloud storage services to store, edit and retrieve their data stored in remote servers and which can be accessed all over the Internet. Such systems are hosted by cloud-based datacenters spread all over the world and are generally equipped with a set of features that allow sharing and collaboration between the users. That is why these popular applications account for a major share of Internet traffic today [49].

Small and medium-sized personal clouds with limited budget constraints generally have fixed amount of bandwidth. This bandwidth is shared by all the concurrent active end-users, which might jeopardize the overall quality of service especially when the demand becomes high. As a matter of fact, these systems are based on a client-server architecture and the default content distribution protocol is usually HTTP. This means that all download requests are handled by a central entity which sends the requested content in a single stream. Unfortunately, such transfer is limited by the narrowest network condition along the way, or by the server being overloaded by requests from many clients.

To cope with these limitations, the cloud can benefit from the clients' upload capacities to overcome its bandwidth limits. This can be done by using the BitTorrent (BT) protocol [50] to distribute the files that are shared between a set of devices. In such scenarios, it is possible to benefit from the common interest of users in the same file and use their own upload bandwidth to offload the cloud from doing all the serving. The two following common file distribution scenarios could benefit from our hybrid download strategy:

1. Synchronization: User A is adding a new file f to his personal account. During the synchronization process, the same file will be download by all the other synchronized devices of the user (figure 7a)
2. Sharing: User A is sharing a file f with other users. In this case, the file will be downloaded by all the synchronized devices of the users (Figure 7b)



(a) Synchronization (b) Sharing

Figure 7: Synchronization and sharing in personal cloud systems

However, the use of BitTorrent may incur a longer download time compared to HTTP especially for small files [2]. The main challenge is to decide for each swarm which protocol is more suitable (HTTP or BT) for transferring the requested files and how much cloud bandwidth should be allocated to each swarm.

5.1 Architecture

Figure 8 presents the general architecture of a PC. This figure is inspired from the official architecture of Dropbox [51]. It presents the core elements of a PC, without taking into consideration the authentication and encryption layers that are deployed to reinforce security. These elements are:

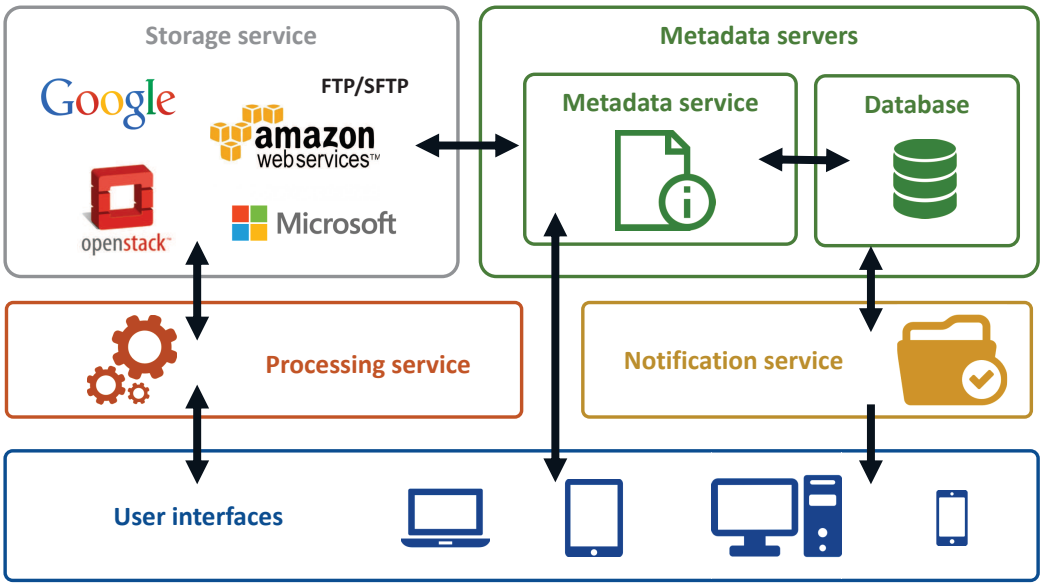


Figure 8: General architecture of personal clouds

- Meta-data service: The meta-data servers contain all the meta-data information related to the clients and the files. They can be equipped with a local database where all the meta-data is stored.
- Storage service: The storage service of storage back-end refers to the physical locations where the users' file content are stored. It can be local, in the form of local storage servers accessed via FTP/SFTP, or external, provided by a third-party (Amazon, Google...).
- Notification service: The notification service is dedicated to monitoring whether or not any changes have been made to the users' accounts. Whenever a change to any file takes place, the client is notified in order to synchronize these changes.
- PC clients or user interfaces: The services offered by personal clouds can be utilized and accessed by physical clients through a number of interfaces, including web interfaces (accessed through web browsers), desktop applications or mobile apps.
- Processing service: The processing service is responsible for processing the files and ensuring their delivery to the end-users. To download a file, the client sends an HTTP GET request to the processing service. The latter verifies the existence of the file in the storage nodes and the file is transferred using the HTTP protocol.

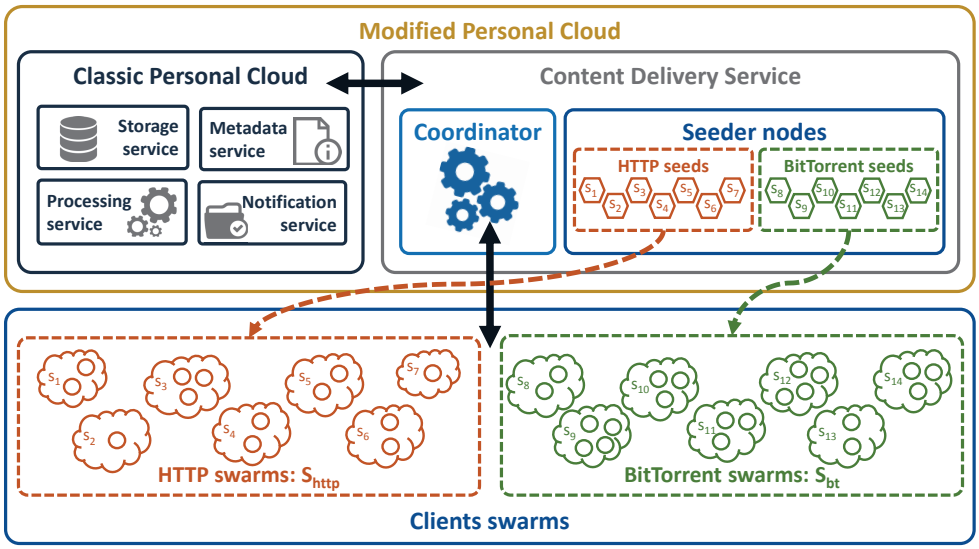


Figure 9: Global view of the system architecture with BT components

To allow inter-client content transfers via BT, several components are added to accommodate the BT behavior. These components are shown in Figure 9 and include:

- Content Delivery Service : The content delivery service is also referred to as cloud. Its main role is to process the requests coming from the end-users and ensure the delivery of the files to the corresponding requesters. Several components are added, compared to the default architecture (Figure 8), including:
  - Coordinator: The coordinator is the core component of the cloud. It is responsible for managing all the clients' requests and ensuring they are processed correctly. The coordinator is also responsible for the proper management of the cloud's resources.
  - Seeder nodes: The seeder nodes are the entities responsible for delivering the requested content from the storage back-end servers to the end-users. To each file being distributed corresponds one seeder node. In our paper, we refer to these seeder nodes as *cloud seeds* or *seeds*. We distinguish two types of seeds: *HTTP seeds* and *BitTorrent seeds* depending on the algorithm adopted to distribute the requested content to end-users.
- Clients swarms: All the end-user peers are organized into swarms. We define a swarm by the set of peers that are requesting the same file. If a file is being downloaded by a single peer, we consider it as a single-peer swarm. This means that, at a given time, there are as many swarms as the number of files being downloaded (to each file corresponds only one swarm and one seeder node). In our model, we distinguish between two types of swarms:
  - HTTP Swarms: The HTTP swarms are the swarms whose peers are downloading the corresponding file from HTTP seeds via HTTP. Clearly, these peers are not collaborating with each other, but grouping them in swarms is a simple means of control which will help, later on, in making the switching decision.

- BitTorrent Swarms: Also referred to as *BT swarms*. Similar to HTTP swarms, BT swarms are the swarms whose peers are downloading the corresponding file from BT seeds via the BT protocol. Typically, these swarms are composed of two peers or more. Since the peers are supposed to collaborate between each other with the help of the cloud seed, it makes no sense to have a single-peer BT swarm.

To download a file, the client sends an HTTP GET request to the coordinator. The latter verifies the existence of the file in the storage nodes and decides the download protocol to be used: HTTP or BitTorrent. The decision is made based on the load on the seed and the swarms' characteristics. In the case of a HTTP download, a HTTP seeder node is associated with the requested file which will be transferred using the HTTP protocol. Otherwise, in the case of a BT transfer, the coordinator creates a torrent meta-data file and runs a corresponding BT seed. After that, the recently created *.torrent* file will be transmitted to the corresponding clients who, unaware of all these interactions, will then start downloading the file using the BitTorrent protocol (from the cloud seed and/or from the other clients). Evidently, the "old" clients who arrived before the switch to BitTorrent will also benefit from the switch if they did not finish the download. In fact, when an "old" client requests a new part of the file to be downloaded, he will realize that the transfer protocol has changed and will automatically adapt to the new one. Thus, each "old" client will join the swarm with the pieces he already has, which means that he will be probably contributing to the swarm as soon as he switches to BitTorrent in a very transparent way.

5.2 When to switch to BitTorrent?

The main challenge in adopting two different download protocol lies in the choice of which protocol to use for each swarm. The main idea is to switch from HTTP to BitTorrent upon detection of an increasing number of requests on a specific content. While this approach seems to be very convenient for big files, it might incur a significant increase in download time for the small ones [2].

To this extend, it is important to identify the best switching point that will help avoiding bottlenecks without affecting significantly the download time. There are many important parameters that should be considered when choosing this point, including: the size of the shared file, the bandwidth of the cloud allocated to that file, the number of peers downloading the file and their corresponding bandwidth capacities. To this extent, the choice of the switching point should be based on a complete comparative study of BitTorrent and HTTP in order to determine the most convenient one in each specific case. This study should be able to answer the following question: *How much time would the clients gain (or lose) and how much bandwidth could the cloud save, if the download protocol is switched from HTTP to BitTorrent?*

To compare the efficiency of , we consider the case of a swarm s composed of L_s distinct peers requesting the same file f_s from the same source, called the cloud seed (or simply the seed). We denote by w_s the allocated share of the cloud's upload bandwidth reserved to s, F_s the size of the shared file f_s, u_s the average upload speed of the peers in the s and by $d_{min,s}$ the download speed of the slowest peer among them (see Figure 10 for more details).

In the following subsections, we will present some formulas related to the estimation of the download times in HTTP and BT (respectively T_{http} and T_{bt}. We will also define the gain

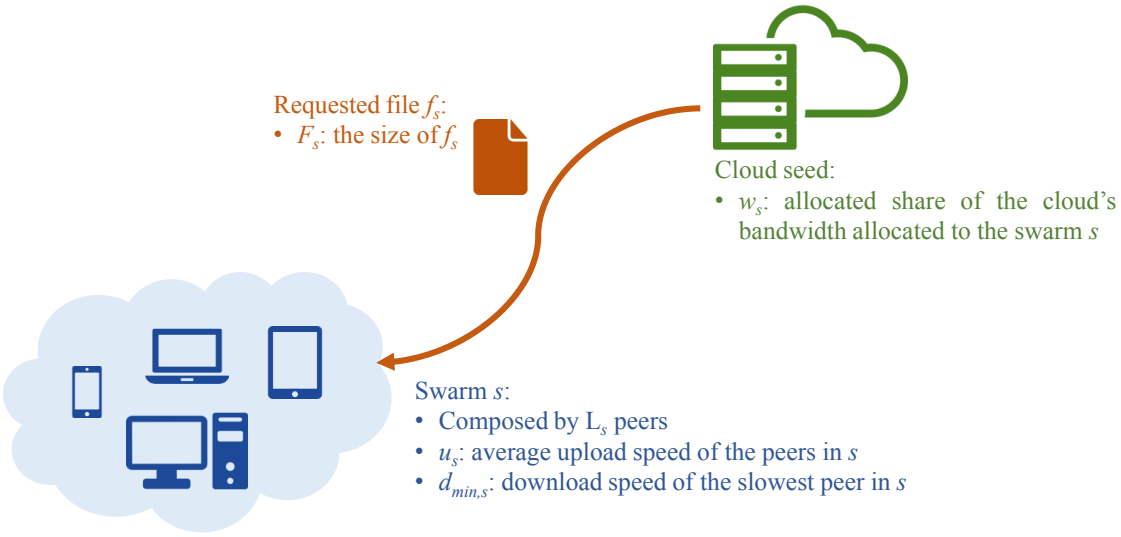


Figure 10: File distribution scenario

percentage *Gain* and estimate the amount of bandwidth offloaded *Offload*.

5.2.1 The distribution time for small files in BitTorrent

To get an estimation of the download time in BitTorrent-like systems, we borrow the following formula proposed in [53] by Kumar and Ross:

$$T_{pa}\left(w_s, s\right) = \frac{F_s}{\min\left\{d_{min,s}, \frac{w_s + L_s\, u_s}{L_s}, w_s\right\}}, \tag{1}$$

where $T_{pa}\left(w_s, s\right)$ is the minimum time needed to distribute a file of size F_s to the L_s leechers in s. This time depends on the download speed of the slowest peer $d_{min,s}$, the aggregated upload bandwidth of all the nodes divided equally between all the L_s leechers, and the upload bandwidth of the cloud seed(s). The authors presented in their paper a complete proof of the download time. The proof is organized into the following exhaustive cases depending on the parameter that may be responsible for the transfer bottleneck:

1. Case A: $d_{min,s} \leq \min\left\{\frac{w_s + L_s\, u_s}{L_s}, w_s\right\}$ and $d_{min,s} \leq \frac{w_s + L_s\, u_s}{L_s - 1}$:
   In this case, the download speed of the peers is limited by the download bandwidth of the slowest peer in the swarm $d_{min,s}$.

2. Case B: $d_{min,s} \leq \min\left\{\frac{w_s + L_s\, u_s}{L_s}, w_s\right\}$ and $\frac{w_s + L_s\, u_s}{L_s - 1} \leq d_{min,s}$:
   In Case B, the transfer is limited by the maximum speed at which a leecher can get data from the other leechers, that is $\frac{w_s + L_s\, u_s}{L_s - 1}$.

3. Case C: $\frac{w_s + L_s\, u_s}{L_s} \leq \min\left\{w_s, d_{min,s}\right\}$:
   The transfer bottleneck in this case is limited by the aggregated upload speed of the network $\left(w_s + L_s\, u_s\right)$ divided equally between the L leechers.

Table 5.2a Table of notations

Symbol	Meaning
W	the cloud's upload budget limit
S	the set of all active swarms
S_{http}	a subset of S that corresponds to the set of the swarms downloading the files via HTTP
S_{bt}	a subset of S that corresponds to the set of the swarms switched to BitTorrent
s	a swarm $s = (P_s, f_s, w_s, isBT_s)$ is identified by the set of the peers forming it P_s, the file being shared f_s, the corresponding amount of allocated cloud bandwidth w_s and a boolean variable $isBT_s$ that indicates the download protocol.
P_s	set of all the peers in s. $P_s = \{(u_p, d_p),\ \forall p \in P_s\}$ where u_p and d_p are respectively the upload and download speeds of a given peer $p \in s$
f_s	file requested by the peers in P_s
w_s	amount of cloud bandwidth allocated to the swarm s
$isBT_s$	boolean variable that indicates the download protocol adopted by the peers in s. $isBT_s = True$, if peers in P_s are downloading f_s via BitTorrent and $isBT_s = False$, otherwise
F_s	size of the requested file f_s
L_s	number of peers in $P_s s$ ($L_s = \|P_s\|$)
$d_{min,s}$	the download speed of the slowest peer in P_s ($d_{min,s} = \min_{\forall p \in P_s} d_p$)
u_s	the average upload speed of all the peers in P_s ($u_s = \sum_{p \in P_s} \frac{u_p}{L_s}$)
η_s	the effectiveness of file sharing, introduced in [52] and reused in [2]. η_s takes real values in $[0,1]$ where 1 means maximum effectiveness while a value of 0 signals the absence of collaboration between peers
α_{bt}	the overhead related to the start-up phase in BitTorrent transfers
τ	the QoS constraint that defines the switching point from HTTP to BT

4. Case D: $w_s \leq \min\left\{d_{min,s}, \frac{w_s + L_s\, u_s}{L_s}\right\}$:
   In this case, the upload bandwidth of the seed w_s is the maximum limit at which each peer can download "fresh" content.

For each of the cases listed above, the authors in [53] constructed a seeding rate profile $s_i(t)$ which denotes the bit rate at which the seeds send pieces to leecher i at time t.
The adopted distribution scheme is the following: As soon as a leecher li begins to receive data from the seed, it replicates it to each of the other $(L_s - 1)$ leechers at a rate $x_i(t)$, where $x_i(t) \leq s_i(t)$, as shown in Figure 11. For each case, the distribution scheme consists of L application-level multicast trees, each rooted at a specific seed, passing through one of the leechers and terminating at each of the $(L_s - 1)$ other leechers.

To calculate the offload ratio in the following section, we need to measure the volume of data offloaded from the cloud. We present here the seeding rate for each case. This rate, denoted by $s_i(t)$ for the sake of clarity, depends on the time t, the file size F_s, the upload speed of the seeds w_s, and the set of upload and download speeds of all the leechers in s. For a complete proof and more details regarding these formulas, we kindly refer the reader

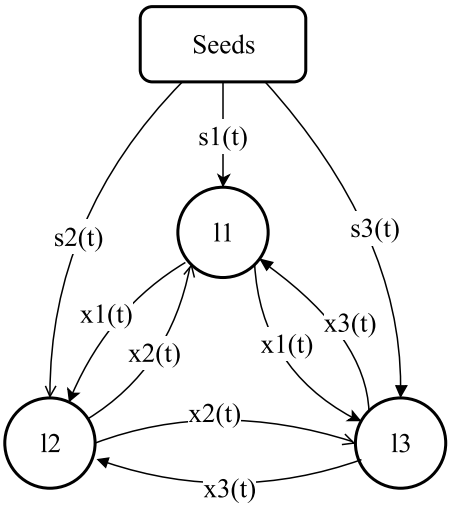


Figure 11: General distribution scheme structure: Leecher li ($i \in \{1,2,3\}$) downloads "fresh" data at the rate $s_i(t)$ from the seeds. The data is replicated later to the other 2 leechers at a rate $x_i(t) < s_i(t)$.

to the original paper [53].

$$s_i(t) = \begin{cases} \frac{u_i\, d_{min,s}}{L_s\, u_s} & \underline{Case\ A} \\ \frac{u_i - L_s\, u_s}{L_s - 1} + d_{min,s} & \underline{Case\ B} \\ \frac{u_i - L_s\, u_s}{L_s - 1} + \frac{w_s + L_s\, u_s}{L_s} & \underline{Case\ C} \\ \frac{u_i\, w_s}{L_s\, u_s} & \underline{Case\ D} \end{cases} \tag{2}$$

Adding the BitTorrent overheads One of the limitations of equation (1) is that it does not take into consideration the overhead that peer-assisted systems may present compared to the client-server ones. These overheads may be neglected for large files. However, they cannot be ignored for the small ones, for which the download time is in the order of a few seconds.

To illustrate the important role that this overhead plays in the distribution of small files, we ran several experiments distributing a 1MB file to several clients. We measured the experimental download times and compared them to the estimated ones using (1). We calculated also the absolute and relative errors. We group all these results in Table 5.2b, where the estimated and experimental download times, and the absolute error are all measured in seconds.

As we can see in Table 5.2b, the difference between the estimated and experimental results can exceed 50% in some cases, which proves that an accurate estimation should include the protocol overheads. These overheads can be mainly of two types, each related to a different phase of BitTorrent:

- Overhead related to the start-up phase: Before starting the download, there are a few steps that each leecher needs to perform: First, the leecher has to get and read the

Table 5.2b Estimated versus experimental distribution time with BitTorrent of a 1MB file. The seed bandwidth is limited to 5 Mbps and the clients are homogeneous each having an upload and download speed of respectively 1 and 2 Mbps.

Clients count	2	3	4	5
Estimated time	4 s	4 s	4 s	4 s
Experimental time	5.51 s	5.47 s	6.03 s	6.25 s
Absolute error	1.51 s	1.47 s	2.03 s	2.25 s
Relative error	37.75%	36.75%	50.75%	56.25%

.torrent file that contains all the meta-info data about the requested content. And then, it needs to contact the tracker(s) to get a list of other peers sharing or downloading the same file. After locating and connecting to the peers, the leecher can finally begin the transfer.
This overhead is relative to the architecture of the system. It can be monitored and dynamically adapted based on the load of the system. We experimentally studied this overhead and noticed that it can be simply modeled as a constant duration α_{bt} added to the download time. More details about the experimental evaluation of α_{bt} can be found in [2].

- Overhead related to the download phase: In BitTorrent, peers upload to each other even though they may only have parts of the file. This can result in upload interruptions when the uploader has no pieces to offer to his unchoked peers.
  Fortunately, this problem has already been tackled in [52], where the authors introduced a parameter to scale down the upload speed of leechers. This parameter, denoted as $\eta \in [0,1]$, measures the effectiveness of file sharing. It can be computed as follows:
  $$\eta = 1 - \mathbb{P}\left\{ \begin{array}{c} \text{downloader } i \text{ has no piece that} \\ \text{his unchoked peers need} \end{array} \right\}$$
  The authors derived this probability and came to the conclusion that η can be expressed as: [17]
  $$\eta_s = 1 - \sum_{n_i=0}^{N-1} \frac{1}{N} \left(\frac{N - n_i}{N\,(n_i + 1)} \right)^k$$
  where N is the number of pieces of the served file and k the number connections a peer has.
  The authors in [52] focused on the case of large files and concluded that $\eta \approx 1$ when N is high. Let us now consider a small file of 1MB composed of $k = 4$ chunks each of 256KB. For $N = 2$, the above equation yields $\eta_s = 0.7069$, which means that there is a probability of about 30% that a peer has no pieces for its unchoked peers. This can affect the download time and make it relatively longer. Thus, this overhead should be also considered when estimating the download time in BitTorrent.

Considering the above listed overheads, we were able to extend Eq. (1) in order to provide

[17] The rectified version of [52] which contains the correct expression of η_s can be found at: `http://users.encs.concordia.ca/~dongyu/paper/bittorrent.pdf`

an accurate estimation of the download time in BitTorrent as follows:

$$\boxed{T_{bt}\left(w_s, s\right) = \frac{F_s}{\min\left\{d_{min,s}, \frac{w_s+\eta_s\, L_s\, u_s}{L_s}, w_s\right\}} + \alpha_{bt}} \tag{3}$$

5.2.2 Gain Ratio

To measure the difference between the download times of client-server and peer-assisted systems, we introduce the gain ratio as follows:

$$Gain\left(w_s, s\right) = \frac{T_{cs}(w_s, s) - T_{bt}(w_s, s)}{T_{cs}(w_s, s)}$$

where T_{cs} is the distribution time in a client-server architecture. T_{cs} is limited by the download speed of the slowest peer $d_{min,s}$ or the bandwidth of all the seeds w_s divided equally between the L_s clients. T_{cs} can be simply defined as follows:

$$T_{cs}\left(w_s, s\right) = \frac{F_s}{\min\left\{d_{min,s}, \frac{w_s}{L_s}\right\}} \tag{4}$$

Clearly, the gain can take negative or positive values and can be also equal to zero. For instance, if the gain is positive, this means that downloading the file via BitTorrent takes less time than using HTTP. To derive the equation of the gain, we distinguish four different cases based on the values of $\min\left\{d_{min,s}, \frac{w_s+\eta_s\, L_s\, u_s}{L_s}, w_s\right\}$ and $\min\left\{d_{min,s}, \frac{w_s}{L_s}\right\}$:

1. Case I: $d_{min,s} \leq \frac{w_s}{L_s}$ and $d_{min,s} \leq \min\left\{\frac{w_s+\eta_s\, L_s\, u_s}{L_s}, w_s\right\}$:
   In this case, the bottleneck in HTTP and BitTorrent is the download speed of the slowest peer. The corresponding download times are: $T_{cs} = \frac{F_s}{d_{min,s}}$ and $T_{bt} = \frac{F_s}{d_{min,s}} + \alpha_{bt}$.

2. Case II: $\frac{w_s}{L_s} \leq d_{min,s}$ and $d_{min,s} \leq \min\left\{\frac{w_s+\eta_s\, L_s\, u_s}{L_s}, w_s\right\}$:
   In Case II, the bottleneck in HTTP is $\frac{w_s}{L_s}$, while it is equal to $d_{min,s}$ in BitTorrent. The corresponding download times are: $T_{cs} = \frac{F_s\, L_s}{w_s}$ and $T_{bt} = \frac{F_s}{d_{min,s}} + \alpha_{bt}$.

3. Case III: $\frac{w_s+\eta_s\, L_s\, u_s}{L_s} \leq \min\left\{d_{min,s}, w_s\right\}$:
   In this case, the bottleneck in BitTorrent is $\frac{w_s+\eta_s\, L_s\, u_s}{L_s}$. And since $w_s \leq w_s + \eta_s\, L_s\, u_s$ and $\frac{w_s+\eta_s\, L_s\, u_s}{L_s} \leq d_{min,s}$, this means that $\frac{w_s}{L_s}$ is always $\leq d_{min,s}$. Thus, in this case, $T_{cs} = \frac{F_s\, L_s}{w_s}$ and $T_{bt} = \frac{F_s\, L_s}{w_s+\eta_s\, L_s\, u_s} + \alpha_{bt}$.

4. Case IV: $w_s \leq \min\left\{d_{min,s}, \frac{w_s+\eta_s\, L_s\, u_s}{L_s}\right\}$:
   Since $\frac{w_s}{L_s} \leq w_s$ and $w_s \leq d_{min,s}$, this means that $\frac{w_s}{L_s}$ is always $\leq d_{min,s}$. In this case, $T_{cs} = \frac{F_s\, L_s}{w_s}$ and $T_{bt} = \frac{F_s}{w_s} + \alpha_{bt}$.

For each of the previous cases, we substitute T_{cs} and T_{bt} to derive the gain ratio as follows:

$$Gain\,(w_s, s) = \begin{cases} -\dfrac{\alpha_{bt}\, d_{min,s}}{F_s} & \underline{Case\ I} \\ 1 - \dfrac{w_s}{L_s\, d_{min,s}} - \dfrac{\alpha_{bt}\, w_s}{F_s\, L_s} & \underline{Case\ II} \\ 1 - \dfrac{w_s}{w_s + \eta_s\, u_s\, L_s} - \dfrac{\alpha_{bt}\, w_s}{F_s\, L_s} & \underline{Case\ III} \\ 1 - \dfrac{1}{L_s} - \dfrac{\alpha_{bt}\, w_s}{F_s\, L_s} & \underline{Case\ IV} \end{cases} \tag{5}$$

5.2.3 Offload Ratio

The offload ratio defines the amount of data offloaded from the cloud seed. It is determined by the total amount of data exchanged between the peers divided by the total downloaded data volume:

$$Offload\,(w_s, s) = \frac{data\ from\ peers}{total\ data\ sent} = 1 - \frac{data\ from\ cloud}{total\ data\ sent} = 1 - \frac{\sum_{i \in \mathcal{L}} \int_0^{T_{bt}(w_s,s)} s_i(t)dt}{F_s\, L_s}$$

where $s_i(t)$ is the seeding rate. Taking into consideration the seeding rate as defined in (2), we can deduce the offload rates as follows:

$$Offload\,(w_s, s) = \begin{cases} 1 - \frac{1}{L_s} & \underline{Case\ A} \\ \frac{\eta_s\, L_s\, u_s}{L_s\, d_{min,s}} & \underline{Case\ B} \\ 1 - \frac{w_s}{w_s + \eta_s\, L_s\, u_s} & \underline{Case\ C} \\ 1 - \frac{1}{L_s} & \underline{Case\ D} \end{cases} \tag{6}$$

5.2.4 The quality of service constraint τ

We presented in the previous sections two key parameters that can help us measure the tradeoff between HTTP and BitTorrent. The gain ratio measures the gain or loss in time that the leechers might experience when switching from HTTP to BitTorrent. The offload ratio gives an estimation of the amount of data that can be offloaded from the server thanks to BitTorrent.

It is clear that if we neglect a potential increase in download time caused by the switch to BitTorrent, the overall offload ratio will always be the highest possible. However, it is equally important to not degrade significantly the download service for the clients. We distinguish the four following cases based on the constraints that can be placed on these parameters:

i. The first possible solution is to put no constraints, that is, BitTorrent is always used when the number of leechers $L_s \geq 2$. In this case, the overall offload ratio will be the highest possible. But, the clients might experience a longer download time.

ii. Another possible solution is to put a limit on the offload ratio: the cloud switches to BitTorrent only when the offload is important. For example, the cloud can decide to switch only when the estimated offloaded bandwidth is above 50% of the total bandwidth, regardless of the download time.

iii. The third possible case is fixing a gain limit: the cloud decides to switch only when the download time in BitTorrent compared to HTTP does not exceed a certain threshold. This threshold can be put on the gain ratio to ensure a minimal bound on the permitted loss in download time.

iv. The last possibility is fixing both the gain and offload ratios. While this case presents an efficient strategy to avoid unnecessary switches, it might be too strict and could limit the overall offload ratio.

After listing all the possible scenarios, we believe that the most convenient procedure to manage the download protocols is the third one. To this extent, we pose τ as the gain constraint. If $\tau \leq 0$, it means that the system tolerates a potential increase in the download time that could occur because of the switch. However, a positive value of τ reflects a stricter constraint. For instance, $\tau = -0.5$ means that an increase up to 50% of the download time is tolerated. Note that a constraint of this magnitude is possible, because $\tau = -0.5$ could represent, for small files, a slight increase in the download time, in the order of a few seconds, to be more precise.

τ can take different values depending on the type of the user account. The choice of its concrete value is left up to the system administrator depending on his needs. A possible concrete example of τ is the following: Suppose that a given service provider cannot gain in bandwidth at the expense of worsening the download time for premium users who are those who are paying money for the service. For this type of clients, τ should be always ≥ 0. However, for free users, which represent a significant portion of the overall user mass[18], it is possible to loosen that constraint, and tolerate delays of up to 50% (which corresponds to $\tau = -0.5$), for instance.

5.2.5 Solving the equation $Gain(w_s^{bt}, s) \geq \tau$

In order to calculate the minimum amount of cloud bandwidth needed to ensure that the switching condition $Gain(w_s^{bt}, s) \geq \tau$ is satisfied, it essential to reverse the gain formulation (equation 5). To this extend, we study the behavior of the gain formulas when w_s^{bt} varies. Based on this constraint, we identify two exhaustive cases in which the gain equations are monotonically decreasing. For each case, we deduce the reversed equations of the gain, interval per interval, as follows[19]:

- **Case A:** $(L_s - 1)\, d_{min,s} \;\geq\; L_s\, \eta_s\, u_s$: the average download speed of the peers in the

[18] 96% of Dropbox clients use the free version of the service (Souce: `http://www.economist.com/blogs/babbage/2012/12/dropbox`)

[19] For the complete proof of the solution, please refer to [54]

swarm s is higher than the upload bandwidth the whole swarm can provide:

$$w_s^{bt} = \begin{cases} L_s\, d_{min,s}, & \forall \tau \in \left]-\infty, -\frac{\alpha_{bt} d_{min,s}}{F_s}\right] \\ \frac{(1-\tau) F_s\, L_s\, d_{min,s}}{F_s + d_{min,s}\, \alpha_{bt}}, & \forall \tau \in \left[-\frac{\alpha_{bt}\, d_{min,s}}{F_s}, \frac{\eta_s\, u_s}{d_{min,s}} - \frac{\alpha_{bt}\, (d_{min,s} - \eta_s\, u_s)}{F_s}\right] \\ \frac{\sqrt{a^2 b^2 - 2abc + 4ab + c^2} - ab - c}{2\, b}, & \forall \tau \in \left[\frac{\eta_s\, u_s}{d_{min,s}} - \frac{\alpha_{bt}(d_{min,s} - \eta_s\, u_s)}{F_s}, 1 - \frac{1}{L_s} - \frac{\alpha_{bt}\, \eta_s\, u_s}{(L_s - 1)\, F_s}\right] \\ \frac{F_s[L_s\,(1-\tau) - 1]}{\alpha_{bt}}, & \forall \tau \in \left[1 - \frac{1}{L_s} - \frac{\alpha_{bt}\, \eta_s\, u_s}{(L_s - 1)\, F_s}, 1 - \frac{1}{L_s}\right[\\ \nexists, & \forall \tau \in \left[1 - \frac{1}{L_s}, +\infty\right[\end{cases}$$

Where:

$$a = \eta_s\, L_s\, u_s, \quad b = \frac{\alpha_{bt}}{F_s\, L_s} \text{ and } c = \tau \tag{7}$$

- **Case B:** $(L_s - 1)\, d_{min,s} \leq L_s\, \eta_s\, u_s$: the average download speed of the peers in the swarm s is lower than the upload bandwidth the whole swarm can provide:

$$w_s^{bt} = \begin{cases} L_s\, d_{min,s}, & \forall \tau \in \left]-\infty, -\frac{\alpha_{bt}\, d_{min,s}}{F_s}\right] \\ \frac{(1-\tau) F_s\, L_s . d_{min,s}}{F_s + d_{min,s}\, \alpha_{bt}}, & \forall \tau \in \left[-\frac{\alpha_{bt}\, d_{min,s}}{F_s}, 1 - \frac{1}{L_s} - \frac{\alpha_{bt}\, d_{min,s}}{F_s\, L_s}\right] \\ \frac{F_s\, [L_s (1-\tau) - 1]}{\alpha_{bt}}, & \forall \tau \in \left[1 - \frac{1}{L_s} - \frac{\alpha_{bt}\, d_{min,s}}{F_s\, L_s}, 1 - \frac{1}{L_s}\right[\\ \nexists, & \forall \tau \in \left[1 - \frac{1}{L_s}, +\infty\right[\end{cases}$$

5.2.6 The switching algorithm

The main goal of the switching algorithm (Algorithm 1) is to evaluate for each requested file the most suitable content distribution model: client-server or peer-assisted, based on the current demand load. Each active seeder node in the system is associated with a swarm of clients that are interested in the same file. It is important to remind here that the default bandwidth distribution protocol is HTTP, but BitTorrent can be also used when the switching conditions previously stated are satisfied. The swarms whose peers are using HTTP as a transfer protocol are referred to as *HTTP swarms* (S_{http} is the set of HTTP swarms) and the ones with peers downloading via BT are labeled as *BT swarms* (S_{bt} is the set of BT swarms).

Algorithm 1 Switching algorithm

Input p^* ▷ the new coming peer
Input s^* ▷ swarm to which p^* belongs
Input w_s^* ▷ seed bandwidth allocated to s^*
Input τ ▷ the switching constraint

1: **if not** $isBT_{s^*}$ **then** ▷ s^* is a HTTP swarm
2: calculate $Gain(w_{s^*}, s^*)$
3: **if** $Gain(w_{s^*}, s^*) \geq \tau$ **then** ▷ switching to BT
4: create a *.torrent* file for f_{s^*}
5: launch a BT seed for f_{s^*} in the cloud
6: **for all** $p \in s^*$ **do** ▷ for all peers in s^*, including p_{s^*}
7: send the *.torrent* to p
8: launch a BT leecher inside p
9: start BT transfer
10: **end for**
11: $isBT_{s^*}$ =**True**
12: **else**
13: p^* downloads the file via HTTP
14: **end if**
15: **else** ▷ s^* is already a BT swarm
16: send the *.torrent* to p^*
17: launch a BT leecher inside p^*
18: **end if**

The algorithm is executed whenever a new peer p^* joins a swarm $s^* \in S$, to download a file f_{s^*}. If the file is already requested by other peers, then p^* will be added to the existing swarm s^*. Otherwise, a new swarm s^* will be created containing a single peer p^* in s^* changes. Based on the protocol already being used for s^*, the switching algorithm works as follows:

- If the peers in s^* are download f^* via HTTP, then the system evaluates the benefit that can be driven from the switch. To do so, it is essential to compute the estimated gain and compare it with the switching constraint τ.
  - When the resulting gain is greater than τ, it means that the quality of service constraint is verified. In this case, the distribution protocol will be switched to BitTorrent: A *.torrent* file corresponding to f_{s^*} will be created and sent to all the peers in s^*. In parallel, a seed will be launched in the cloud. Upon the reception of the *.torrent* file, a BitTorrent leecher will be launched inside each peer in s^*. After this phase, these peers will start downloading the file in BitTorrent, while offloading the cloud from doing all the serving.
  - When the gain is lower than τ, it means that the switching constraints are not verified. In this case, the download protocol is kept unchanged (no switching to BT) and p^* will get the file directly from the cloud seed.
- In the case where s^* is already a BT swarm (before the arrival of p^*), then the new peer will download the corresponding *.torrent* file and join the other peers in s^* to download f_{s^*} via BT

Validation of the switching algorithm To validate the switching algorithm, we validated it on the UB1 trace. We fixed the upload speed of the seed w_s^* to 2 Mbps. We remind our reader that w_s^* does not refer to the total upload bandwidth of the cloud, but to the portion of its bandwidth allocated to the each specific file/swarm. We considered the case were all the clients were homogeneous and have an upload and download speed of 512 Kbps and 1 Mbps, respectively.

We went through the trace focusing on the files that have been downloaded more than once. Our goal was to identify the files with collapsing download times which are the candidates for the switch to BitTorrent. In other words, for each file, we checked if there were consecutive download operations (at time stamps t_1 and t_2) that came before the end of the theoretical download time in HTTP: $t_2 - t_1 \leq T_{cs}$. T_{cs} is calculated based on the settings listed above. After the identification of these files, we calculated for each case the gain ratio using (5). Depending on the gain value and the τ constraint, we identified the files that were subject to switching and measured the corresponding offloaded volume of data using (6).

Table 5.2c Offloaded volume and offload percentage resulting from the application of Algorithm 1 using different τ values

Constraint	Offloaded Volume	Overall Offload%
$\tau = -1.0$	207.35 GB	16.7183%
$\tau = -0.5$	207.33 GB	16.7170%
$\tau = -0.2$	207.04 GB	16.6938%
$\tau = 0.0$	137.64 GB	11.0979%
$\tau = 0.2$	137.59 GB	11.0942%
$\tau = 0.5$	90.60 GB	7.3055%
$\tau = 1.0$	0.0 GB	0.0%

Table 5.2c presents the results of the application of Algorithm 1 on the trace. The overall offload percentage is calculated based on the percentage ratio between the offloaded volume and the total downloaded volume (1,240.25 GB). We varied the values of the switching constraint τ in order to get a global idea of the gains, and we noticed that if we fixed τ to tolerate losses of 20% ($\tau = -0.2$), the cloud load could be reduced up to 16%. In the case of stricter constraints, e.g., no loss is tolerated ($\tau = 0$), or no switch unless we gain 20% in download time ($\tau = 0.2$), the overall offload percentage falls down to around 11%.

Even though the UB1 system is not very popular, our algorithm could achieve savings up to 16% in terms of cloud bandwidth. We strongly believe that this offload would be higher on other systems, like Dropbox or Google Drive, which have more users and more file sharing.

To measure the amount a money that can be saved using our algorithm, we consider a cloud storage system that uses Amazon Simple Storage Service (S3) as a storage back-end. At the time of writing this paper, the standard charging rates for data transfer were[20]:

- \$0.0 per GB for the first 1 GB/month
- \$0.12 per GB for transfers up to 10 TB/month

[20]More information about the complete and updated rates can be found at `http://aws.amazon.com/s3/pricing/`

- $0.09 per GB for the next 40 TB/month

Using these rates, the overall data transfer cost is approximately $\$3,000$ per month. Fixing the gain constraint to $\tau = -1$ would lead to savings of about $450 per month which is about $\$5,374$ per year. These savings will be higher for systems that involve more sharing than UB1.

5.2.7 The bandwidth allocation algorithm

In this section we present our bandwidth distribution and protocol management algorithm. This algorithm aims to minimize the cloud's allocated bandwidth among the seeder nodes, while respecting the QoS constraint. We remind that a seeder node is an entity responsible for distributing a given file to the corresponding set of clients.

In addition to the bandwidth allocations, the algorithm is also responsible for evaluating for each swarm the most suitable content distribution protocol: HTTP or BitTorrent. A swarm would switch to BitTorrent if it satisfies the following conditions:

1. The number of clients in the swarms is higher or equal to 2. In fact, it makes no sense to use BitTorrent with only one client interested in the file.
2. The switch to BitTorrent should satisfy the quality of service constraint τ. This means that BitTorrent can be used only when the gain percentage (equation 5) is higher or equal than τ.
3. The amount of cloud bandwidth allocated in BitTorrent should be smaller than the one using HTTP. This means that the switch will only take place if the cloud would gain in terms of bandwidth.

The main goal of our bandwidth distribution and switching algorithm is to optimally manage the cloud's limited bandwidth among the seeder nodes. It is also responsible for evaluating for each requested file the most suitable content distribution model: client-server or peer-assisted, based on the current demand load. Each active seeder node in the system is associated with a swarm of clients that are interested in the same file. It is important to remind here that the default bandwidth distribution protocol is HTTP, but BitTorrent can be also used when the switching conditions previously stated are satisfied. The swarms whose peers are using HTTP as a transfer protocol are referred to as *HTTP swarms* (S_{http} is the set of HTTP swarms) and the ones with peers downloading via BT are labeled as *BT swarms* (S_{bt} is the set of BT swarms).

The algorithm is executed whenever a change affects a swarm $s^* \in S$. This change can be related to a modification in one or more of the parameters of a certain swarm. It can be due to one or more of the following cases:

- A new peer p^* wants to download a file f_{s^*}. If the file is already requested by other peers, then p^* will be added to the existing swarm s^*. Otherwise, a new swarm s^* will be created containing a single peer p^*.

- A peer p^* leaves a swarm s^*. If p^* was not the only peer in the swarm, then the modified swarm will contain a list of the other remaining peers. If p^* was the last peer in s^*, then s^* will be removed from S.

- The upload or download speed of one or more of the peers in s^* changes.

Algorithm 2 Bandwidth distribution and switching algorithm

Input S ▷ the set of all the current swarms
Input s^* ▷ swarm affected by a change
Input W ▷ the cloud's upload bandwidth budget limit
Input τ ▷ the switching constraint

1: **if** $L_{s^*} = 1$ **then** ▷ s^* is a single-peer swarm
2: $w_{s^*} = D_{s^*}$
3: **else if** $L_{s^*} > 1$ **then** ▷ s^* has more than one peer
4: **if** $s^* \in S_{http}$ **then** ▷ s^* is a HTTP swarm
5: calculate $w_{s^*}^{bt}$ using equation (7)
6: **if** $w_{s^*}^{bt} \leq D_{s^*}$ **then** ▷ switching to BT
7: switch the transfer protocol from HTTP to BT
8: $isBT_{s^*} = True$ ▷ mark s^* as a BT swarm
9: $w_{s^*} = w_{s^*}^{bt}$
10: **else** ▷ not switching to BT
11: $w_{s^*} = D_{s^*}$
12: **end if**
13: **else** ▷ $s^* \in S_{BT}$, s^* is a BT swarm
14: $w_{s^*} = w_{s^*}^{bt}$ calculated using equation (7)
15: **end if**
16: **else** ▷ $L_{s^*} = 0$, s^* no longer exists
17: remove s^* from S
18: **if** $\sum_{s \in S} w_s + w_{s^*} = W$ **then** ▷ the cloud was overloaded
19: **for each** s **in** S_{bt} **do**
20: $w_s = w_s + \frac{w_s}{\sum_{s \in S_{bt}} w_s} w_{s^*}$ ▷ redistribute w_{s^*} to the BT swarms
21: **end for**
22: **end if**
23: **end if**

24: **if** $\sum_{s \in S} w_s > W$ **then**
25: **for each** s **in** S **do**
26: $w_s = \frac{w_s}{\sum_{s \in S} w_s} W$ ▷ scale down all the bandwidth shares
27: **end for**
28: **end if**

The algorithm requires the following input parameters: the set of all current swarms S, the swarm affected by the change s^*, the cloud's upload bandwidth budget limit W and the switching constraint τ. Using these input parameters, the algorithm identifies for each swarm the most suitable download protocol (HTTP or BT) and calculates the amount of bandwidth to be allocated to the corresponding seed, as follows:

- If s^* is a single-peer swarm ($L_{s^*} = 1$), then the cloud allocates to s^* a share of bandwidth equal to its download capacity: $w_{s^*} = D_{s^*}$ (*lines 1 and 2*). In this case, the file will be distributed directly from the cloud seed to the single-peer using HTTP.

- If the number of peers in s^* is strictly higher than 1 (*lines 3 to 12*), then there are two possible cases:

  - If the peers in s^* are using HTTP to download f_{s^*} ($isBT_{s^*} = False$), the algorithm verifies if it is worth it to switch to BT. To do so, $w^{bt}_{s^*}$ is calculated according to equation (7). We remind that $w^{bt}_{s^*}$ measures the amount of seed bandwidth required to verify the quality of service constraint τ when using BT for s^*. The algorithm compares later this bandwidth (w_{s^*}) with the bandwidth allocated by default to the swarm (which is equal to D_{s^*}).
    - If the bandwidth required using BT is smaller than the one allocated by default ($w^{bt}_{s^*} \leq D_{s^*}$), then the download protocol more suitable for s^* is BT (*lines 4 to 9*). In this case, a .torrent file associated to f_{s^*} is created and a BT seed is launched in the cloud. All the peers in s^* have to download the .torrent file recently created and then can start downloading f_{s^*} via BT.
    - If the use of BT requires more bandwidth than HTTP, then it is not worth switching to BT. In this case, the cloud allocates a share of bandwidth equal to D_{s^*} (*line 11*).
  - If s^* has already switched to BT, then the algorithm recalculates $w^{bt}_{s^*}$: the bandwidth needed to maintain the quality of service constraint τ, which represents also the amount of bandwidth allocated to s^*.

- If s^* is an empty swarm ($L_{s^*} = 0$), then the swarm is removed from the swarms' list. If the cloud was overloaded before the removal of s^*, then the amount of bandwidth that was previously allocated to s^* is redistributed among the BitTorrent swarms (*lines 18 to 22*). This will prevent the cloud's bandwidth from being underutilized and will boost the distribution of the files among the BT swarms.

When the number of simultaneous requests becomes high, the seed might be unable to serve all the swarms at their full speed. In such a case, the cloud has to scale down all the bandwidth allocations proportionally to the demand (*lines 24 to 28*).

Validation of the bandwidth allocation algorithm In order to evaluate the performance of the proposed algorithm, we implement two simulators. The first simulates the default behavior of the cloud where all the download requests are treated individually and the files are distributed via HTTP. The second simulates the bandwidth distribution and protocol management algorithm. We compare later the results of both approaches using a trace of a real personal cloud system. We run both simulators with a wide combination of τ and W values and collect the logs of each experiment. Then, we evaluate our algorithm comparing the results with the ones obtained using the default strategy with the same bandwidth limits.

First of all, we run the simulator fixing the upload bandwidth budget at 300 Mbps and varying the switching constraint τ. The goal is to get a first idea of the performance of the algorithm. We measure for each simulation, the download time taken by each operation and compare them to the times measured using the HTTP-only simulator with the same budget

limit. It is important here to note that the download times are measured in seconds with a precision of one millisecond. We classify the operations into three different categories: operations that have gained in download time with the algorithm, operations that experienced losses and operations whose download time is left unchanged for both approaches.

Table 5.2d Percentages of operations with gains and losses in download time resulted by the algorithm compared to pure HTTP use. The cloud upload bandwidth budget limit is W=300 Mbps.

	$\tau_1 = -0.2$	$\tau_2 = 0$	$\tau_3 = 0.2$	$\tau_4 = 0.4$
% of operations with gain	82.89 %	82.9 %	83.43 %	83.53 %
% of operations with loss	2.23 %	2.31 %	2.48 %	2.85 %
% of operations with no difference	14.88 %	14.79 %	14.09 %	13.62 %
Total %	100 %	100 %	100 %	100 %

Table 5.2d presents the percentages of the operations in each category. We notice that for the three different values of τ, more than 80% of the operations benefited from a gain in download time, about 15% kept the same time and only about 2.5% of them lost in download time. Even though these percentages are quite good, we need to make sure that the cumulative gains are higher than the losses. To do so, we sum all the download times of all operations for both approaches and calculate the total net gain percentage (*net_gain_%*). *net_gain_%* represents the percentage ratio between the total time gained (or lost) by using the algorithm (*net_gain_hours* = *sum_http_hours* − *sum_algo_hours*) and the total download times using HTTP only (*sum_http_hours*).

$$net\_gain\_\% = \frac{net\_gain\_hours}{sum\_http\_hours} \times 100 = \frac{sum\_http\_hours - sum\_algo\_hours}{sum\_http\_hours} \times 100$$

Table 5.2e Total sum of all the download times for all the operations and the net gain percentage for the algorithm applied on the one-hour sample of the UB1 trace. The cloud upload bandwidth budget limit is W=300 Mbps.

	$\tau_1 = -0.2$	$\tau_2 = 0$	$\tau_3 = 0.2$	$\tau_4 = 0.4$
sum_http_hours (in hours)	2450.2	2450.2	2450.2	2450.2
sum_algo_hours (in hours)	2000.98	1997.05	1952.73	1906.56
net_gain_hours (in hours)	449.22	453.15	497.47	543.64
net_gain_%	**18.33**%	**18.49**%	**20.3**%	**22.19**%

Table 5.2e presents the total sum of all the download times of all the download operations and the net gain percentage based on the UB1 one-hour sample. The first row represents the sum of download times using HTTP. It is important to mention here that, for HTTP, this sum depends only on the cloud upload bandwidth budget W. Hence, for the fixed bandwidth $W = 300$ Mbps, it is always equal to 2450.2 hours, regardless of the τ constraint. However, the sum of the download times using the algorithm with a given cloud bandwidth limit depends highly on the switching constraint τ. In Table 5.2e, we compare the results with three different τ values: The first constraint is $\tau_1 = -0.2$: this constraint can be translated as follows: at a certain timestamp, a swarm can switch from HTTP to BitTorrent only if it

will only lose less than 20% in download time. Under this constraint, we notice that the algorithm performs better than HTTP with a net gain in the client's download time equal to 18.33%. Next, we make the constraint a little bit stricter and we accept only switches to BT when the peers in question will only gain in download ($\tau_2 = 0$, no loss is permitted). We notice that the net gain percentage improves slightly. This is because the constraint will prevent swarms with negative gains from switching which will result in an increase of the total amount of net gain hours. With the third and fourth constraints $\tau_3 = +0.2$ and $\tau_4 = +0.4$ (switch only if the corresponding peers will gain 20%, respectively 40%, or more gain in download time), the net gain percentage gets higher and reaches more than 20% of the total download time of all peers.

5.3 Implementation: Integration with StackSync

Our hybrid approach can applied widely in any cloud-based system. Personal clouds are the most appropriate for this proposal since the developers can tune the client's implementation to extend them with the BT functionality.

To validate our model, two main components should be added to the StackSync architecture (figure 12): a central coordinator (on the server's side) and a BitTorrent library (on the client's side).

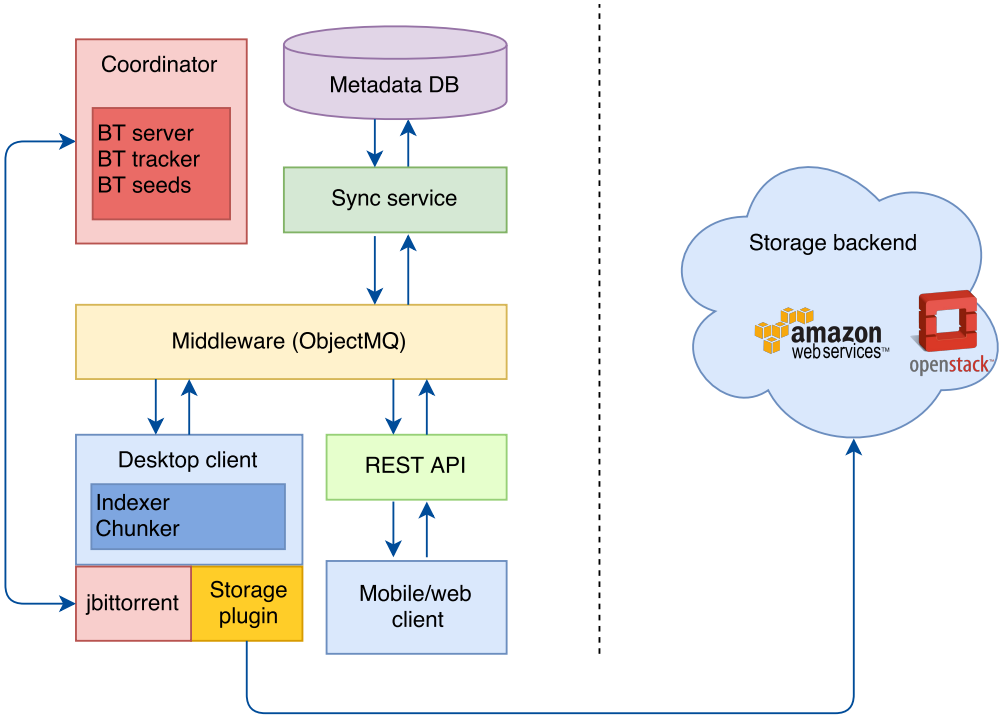


Figure 12: StackSync architecture overview with BitTorrent

The *jbittorrent* [21] library used in our implementation is a simple and easy to use Java implementation of the BitTorrent protocol. It provides a set of classes that allow the creation

[21] jbittorrent is available in GitHub under the GNU GPL license `https://github.com/cloudspaces/jbittorrent`

of the .torrent files and the download and sharing of files between peers in a BitTorrent swarm.

jbittorrent is based on the library Java BitTorrent API[22] developed by Baptiste Dubuis, Artificial Intelligence Laboratory, EPFL. In o *jbittorrent* extends the original library with the following features:

- Optimization of the management mechanism that maintains the connections between the peers of the swarms. The new version of the library solves the multiple connection problem in the version 1.0 of the Java BitTorrent API. This problem was due to the fact that connections are identified by the IP and the port of the peer in question which might result in multiple connections for one peer.
- Improvement of the implementation of the Choking Algorithm by guaranteeing that the number of unchoked peers is exactly five
- Re-implementation of the Optimistic Unchoking process in a way that makes it easier for a peer to get its first pieces
- Implementation of the Rarest First Algorithm in the process of the pieces selection process
- Implementation of the End Game Strategy

We have validated our approach using a trace of the Ubuntu One system [2, 54, 3, 55] and have proven that the used of BitTorrent can reduce significantly the load on the cloud without degrading the download times for the clients. Nevertheless, we have chosen not to include this feature in the final release of StackSync in order to avoid privacy issues.

[22] More information about the Java BitTorrent API can be found at: `http://sourceforge.net/projects/bitext/`

6 Conclusion

In this document we presented several research works that are joined by the same underlying objective: making Personal Cloud storage platforms more secure and convenient.

We analysed the use of erasure coding in a comprehensive solution to handle untrusted and heterogeneous cloud repositories. Further, with the Hybris storage protocol, we have tackled the problem of untrusted storage from another perspective: the challenging integration between private and public clouds, and the related trade-offs on consistency and fault tolerance. Finally, in Section 5, we have presented another interesting case of study on cloud storage. While Hybris approached cloud storage issues from users' point of view, this latter work focused on techniques to exploit peer-to-peer technologies to offload cloud storage platforms and ease files distribution.

We believe that the research contributions presented in this work can match the practical requirements of a real-life Personal Cloud system. We validated this belief by designing and developing solutions that easily integrate with the StackSync prototype.

References

[1] B. Wei, G. Fedak, and F. Cappello, "Scheduling independent tasks sharing large data distributed with bittorrent," in Grid Computing, 2005. The 6th IEEE/ACM International Workshop on, Nov 2005, pp. 8 pp.–.

[2] R. Chaabouni, M. Sanchez-Artigas, and P. Garcia-Lopez, "Reducing Costs in the Personal Cloud: Is Bittorrent a Better Bet?" in Peer-to-Peer Computing (P2P), 14-th IEEE International Conference on, Sept 2014, pp. 1–10.

[3] R. Gracia-Tinedo, Y. Tian, J. Sampe, H. Harkous, J. Lenton, P. Garcia-Lopez, M. Sanchez-Artigas, and M. Vukolic, "Dissecting UbuntuOne: Autopsy of a Global-scale Personal Cloud Back-end," in Proceedings of the 2015 Internet Measurement Conference, IMC 2015, Tokyo Japan, October 28-30, 2014, 2015.

[4] C. Huang, H. Simitci, Y. Xu, A. Ogus, B. Calder, P. Gopalan et al., "Erasure coding in Windows Azure Storage," in Proc. USENIX Annual Technical Conference, 2012.

[5] W. Wong, "Cleversafe grows along with customers' data storage needs," Chicago Tribune, Nov. 2013.

[6] G. R. Goodson, J. J. Wylie, G. R. Ganger, and M. K. Reiter, "Efficient Byzantine-tolerant erasure-coded storage," in dsn04, 2004, pp. 135–144.

[7] C. Cachin and S. Tessaro, "Optimal resilience for erasure-coded Byzantine distributed storage," in dsn06, 2006, pp. 115–124.

[8] J. Hendricks, G. R. Ganger, and M. K. Reiter, "Low-overhead Byzantine fault-tolerant storage," in sosp07, 2007.

[9] P. Dutta, R. Guerraoui, and R. R. Levy, "Optimistic erasure-coded distributed storage," in disc08, ser. lncs, G. Taubenfeld, Ed., vol. 5218. Springer, 2008, pp. 182–196.

[10] A. Bessani, M. Correia, B. Quaresma, F. André, and P. Sousa, "DepSky: Dependable and secure storage in a cloud-of-clouds," in eurosys11, 2011, pp. 31–46.

[11] G. Chockler, R. Guerraoui, I. Keidar, and M. Vukolić, "Reliable distributed storage," IEEE Computer, vol. 42, no. 4, pp. 60–67, Apr. 2009.

[12] M. Vukolić, Quorum Systems: With Applications to Storage and Consensus, ser. Synthesis Lectures on Distributed Computing Theory. Morgan & Claypool, 2012.

[13] M. Herlihy, "Wait-free synchronization," toplas, vol. 11, no. 1, pp. 124–149, Jan. 1991.

[14] M. P. Herlihy and J. M. Wing, "Linearizability: A correctness condition for concurrent objects," toplas, vol. 12, no. 3, pp. 463–492, Jul. 1990.

[15] G. Chockler, R. Guerraoui, and I. Keidar, "Amnesic distributed storage," in disc07, ser. lncs, G. Taubenfeld, Ed., vol. 4731. Springer, 2007, pp. 139–151.

[16] J.-P. Martin, L. Alvisi, and M. Dahlin, "Minimal Byzantine storage," in disc02, ser. lncs, D. Malkhi, Ed., vol. 2508. Springer, 2002, pp. 311–325.

[17] J. Yin, J.-P. Martin, A. V. L. Alvisi, and M. Dahlin, "Separating agreement from execution in Byzantine fault-tolerant services," in sosp03, 2003, pp. 253–268.

[18] I. Abraham, G. Chockler, I. Keidar, and D. Malkhi, "Byzantine disk Paxos: Optimal resilience with Byzantine shared memory," dc, vol. 18, no. 5, pp. 387–408, 2006.

[19] Y. Afek, H. Attiya, D. Dolev, E. Gafni, M. Merritt, and N. Shavit, "Atomic snapshots of shared memory," jacm, vol. 40, no. 4, pp. 873–890, 1993.

[20] J. S. Plank, "Erasure codes for storage systems: A brief primer," ;login: the Usenix magazine, vol. 38, no. 6, pp. 44–50, December 2013.

[21] R. Guerraoui, R. R. Levy, and M. Vukolić, "Lucky read/write access to robust atomic storage," in dsn06, 2006, pp. 125–136.

[22] D. Dobre, M. Majuntke, and N. Suri, "On the time-complexity of robust and amnesic storage," in opodis08, ser. lncs, T. P. Baker, A. Bui, and S. Tixeuil, Eds., vol. 5401. Springer, 2008, pp. 197–216.

[23] C. Cachin, R. Guerraoui, and L. Rodrigues, Introduction to Reliable and Secure Distributed Programming (Second Edition). Springer, 2011.

[24] E. Androulaki, C. Cachin, D. Dobre, and M. Vukolic, "Erasure-coded byzantine storage with separate metadata," CoRR, vol. abs/1402.4958, pp. 1–18, 2014. [Online]. Available: http://arxiv.org/abs/1402.4958

[25] V. R. Cadambe, N. Lynch, M. Medard, and P. Musial, "Coded atomic shared memory emulation for message passing architectures," MIT, CSAIL Technical Report MIT-CSAIL-TR-2013-016, 2013.

[26] D. Dobre, G. Karame, W. Li, M. Majuntke, N. Suri, and M. Vukolić, "PoWerStore: Proofs of writing for efficient and robust storage," in ccs13, 2013.

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

[28] A. N. Bessani, M. Correia, B. Quaresma, F. André, and P. Sousa, "DepSky: Dependable and secure storage in a cloud-of-clouds," ACM Transactions on Storage, vol. 9, no. 4, p. 12, 2013.

[29] 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.

[30] 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.

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

[32] 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.

[33] M. P. Herlihy and J. M. Wing, "Linearizability: A Correctness Condition for Concurrent Objects," ACM Trans. Program. Lang. Syst., vol. 12, no. 3, pp. 463–492, 1990.

[34] M. Herlihy, "Wait-Free Synchronization," ACM Trans. Program. Lang. Syst., vol. 11, no. 1, pp. 124–149, 1991.

[35] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears, "Benchmarking cloud serving systems with ycsb," in SoCC, 2010, pp. 143–154.

[36] M. Pease, R. Shostak, and L. Lamport, "Reaching Agreement in the Presence of Faults," J. ACM, vol. 27, no. 2, pp. 228–234, 1980.

[37] 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.

[38] A. Adya, W. J. Bolosky, M. Castro, G. Cermak, R. Chaiken, J. R. Douceur, J. Howell, J. R. Lorch, M. Theimer, and R. P. Wattenhofer, "Farsite: Federated, available, and reliable storage for an incompletely trusted environment," SIGOPS Oper. Syst. Rev., vol. 36, no. SI, pp. 1–14, Dec. 2002. [Online]. Available: http://doi.acm.org/10.1145/844128.844130

[39] 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. [Online]. Available: http://dx.doi.org/10.1007/s00446-005-0151-6

[40] P. Bailis, A. Fekete, A. Ghodsi, J. M. Hellerstein, and I. Stoica, "The potential dangers of causal consistency and an explicit solution," in ACM Symposium on Cloud Computing, SOCC '12, San Jose, CA, USA, October 14-17, 2012, 2012, p. 22. [Online]. Available: http://doi.acm.org/10.1145/2391229.2391251

[41] L. Lamport, "Time, clocks, and the ordering of events in a distributed system," Commun. ACM, vol. 21, no. 7, pp. 558–565, 1978. [Online]. Available: http://doi.acm.org/10.1145/359545.359563

[42] D. B. Terry, A. J. Demers, K. Petersen, M. Spreitzer, M. Theimer, and B. B. Welch, "Session guarantees for weakly consistent replicated data," in PDIS, 1994, pp. 140–149.

[43] W. M. Golab, X. Li, and M. A. Shah, "Analyzing consistency properties for fun and profit," in PODC, 2011, pp. 197–206.

[44] F. J. Torres-Rojas, M. Ahamad, and M. Raynal, "Timed consistency for shared distributed objects," in PODC, 1999, pp. 163–172.

[45] C. Li, D. Porto, A. Clement, J. Gehrke, N. Preguiça, and R. Rodrigues, "Making geo-replicated systems fast as possible, consistent when necessary," in Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation, ser. OSDI'12. Berkeley, CA, USA: USENIX Association, 2012, pp. 265–278. [Online]. Available: http://dl.acm.org/citation.cfm?id=2387880.2387906

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

[47] D. Dobre, P. Viotti, and M. Vukolić, "Hybris: Efficient and robust hybrid cloud storage," in ACM Symposium on Cloud Computing, SOCC '14, Seattle, WA, USA, November 3-5, 2014, 2014.

[48] P. G. López, M. S. Artigas, S. Toda, C. Cotes, and J. Lenton, "Stacksync: bringing elasticity to dropbox-like file synchronization," in Proceedings of the 15th International Middleware Conference, Bordeaux, France, December 8-12, 2014, 2014, pp. 49–60. [Online]. Available: http://doi.acm.org/10.1145/2663165.2663332

[49] I. Drago, "Understanding and Monitoring Cloud Services." Ph.D. dissertation, University of Twente, 2013.

[50] B. Cohen, "Incentives Build Robustness in BitTorrent," 2003.

[51] Dropbox, Inc., "Dropbox for Business Security: A Dropbox Whitepaper," https://www.dropbox.com/static/business/resources/dfb_security_whitepaper.pdf.

[52] D. Qiu and R. Srikant, "Modeling and performance analysis of bittorrent-like peer-to-peer networks," in Proceedings of the 2004 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications, ser. SIGCOMM '04. New York, NY, USA: ACM, 2004, pp. 367–378. [Online]. Available: http://doi.acm.org/10.1145/1015467.1015508

[53] R. Kumar and K. Ross, "Peer-assisted file distribution: The minimum distribution time," in Hot Topics in Web Systems and Technologies, 2006. HOTWEB '06. 1st IEEE Workshop on, Nov 2006, pp. 1–11.

[54] R. Chaabouni, M. Sanchez-Artigas, and P. Garcia-Lopez, "The Power of Swarming in Personal Clouds Under Bandwidth Budget," to appear.

[55] "Ubuntu One System," http://one.ubuntu.com/.