SEVENTH FRAMEWORK PROGRAMME

CloudSpaces

(FP7-ICT-2011-8)

Open Service Platform for the Next Generation of Personal Clouds

D3.2 Adaptive storage infrastructure

Due date of deliverable: 31-10-2014
Actual submission date: 15-10-2014

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

Summary of the document

Document Type	Deliverable
Dissemination level	Public
State	Final
Number of pages	44
WP/Task related to this document	WP3
WP/Task responsible	EUR
Author(s)	Marko Vukolić, Rahma Chaabouni
Partner(s) Contributing	EUR, URV
Document ID	CLOUDSPACES_D3.2_141015_Public.pdf
Abstract	This document is a report on CloudSpaces adaptive storage layer. The document describes the Cloudspaces approach to distributed erasure-coding techniques and handling untrusted data repositories, in particular in relation to Cloudspaces Hybrid Cloud Storage service (described in D.3.1). The document also discusses the prototype of the adaptive edge platform based on BitTorrent.
Keywords	Cloud storage, synchronization, sharing, interoperability, Personal Cloud

Table of Contents

1 Executive summary

This document is a report on CloudSpaces adaptive storage layer. It describes the Cloudspaces approach to distributed erasure-coding techniques and handling untrusted data repositories, in particular in relation to Cloudspaces Hybrid Cloud Storage service (the early design of which was described in Deliverable 3.1.). The document also discusses the prototype of the adaptive edge platform based on BitTorrent.

2 Introduction

In this deliverable we give an advanced description of the adaptive storage that underlies the Cloudspaces. We pay close attention to the use of erasure coding (to minimize the storage consumption of our solutions), handling untrusted cloud repositories, and adaptivity to different (heterogeneous) data repositories, including different public cloud providers. In particular, we present:

- Wait-free erasure coding over multiple untrusted repositories (AWE). We further 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 AWE in Section 3.

- Hybrid cloud storage (Hybris). Hybris is an advanced version of our initial system described in D3.1, augmented with support for 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. Prototype design of Hybris, designed by EUR, is described in Section 4.

- Adaptive edge platform. We designed an adaptive edge platform that replaces regular HTTP cloud unicasts with BitTorrent. BitTorrent effectively leverages swarms of Multiple personal cloud users to boost the efficiency of personal clouds. The advanced prototype design of Cloudspaces Adaptive edge platform is described in Section 5. This adaptive platform prototype has been massively deployed by two Cloudspaces partners: URV and TISSAT.

3 Erasure-Coded Storage with Separate Metadata

Erasure coding is a key technique that saves space and retains 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 [1, 2] 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 [3, 4, 5, 6, 7], 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 [8, 9]; 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 [3] or rely on node-to-node communication [4], others need computationally expensive public-key cryptography [4, 5] or may block clients due to concurrent operations of other clients [5].

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 [3].

More specifically, AWE is an asynchronous protocol that provides the strongest liveness and safety properties, namely wait-freedom [10] and atomicity (or linearizability) [11]. 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 [12] 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 [13], 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 erasure 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 [14]. The data nodes may be passive objects that support read and write operations but cannot execute code, as in Disk Paxos [15]. 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 [16]. 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.

We continue as follows. The Protocol AWE 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 subection 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 $\mathit{Reconstruct}_{k,n}(F)$ outputs v.

Metadata service. The metadata service is implemented by a standard atomic snapshot object [16], 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 [16], 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 [17, 18].

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 [19], 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 simplified 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 [17, 18], 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 show here for the first time.

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 [20].

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 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. As-

Protocol	**Communication cost**		**Storage cost**
	Write	*Read*	
ORCAS-A [6]	$(1+m)n\ell$	$2n\ell$	$n\ell$
ORCAS-B [6]	$(1+m)n\ell/k$	$2n\ell/k$	$mn\ell/k$
CASGC [21]	$n\ell/k$ *	∞	$mn\ell/k$
CT [4]	$(n+m)n\ell/(k+t)$	ℓ *	$n\ell/(k+t)$ *
HGR [5]	$n\ell/k$ *	∞	$mn\ell/k$
M-PoWerStore [22]	$n\ell/k$ *	$n\ell/k$	∞
DepSky [7]	$n\ell/k$ *	$n\ell/k$	∞
AWE (Sec. 4.3)	$n\ell/k$ *	$(t+k)\ell/k$	$2m^2n\ell/k$

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.

suming 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 [21] and HGR [5], 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 [7] is neither wait-free nor amnesic and M-PoWerStore [22] 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 [1] — as a result, enterprise-class hybrid cloud storage solutions are booming with all leading storage providers, such as EMC[2], IBM[3], Microsoft[4] and others, offering their proprietary solutions.

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 [23]. To this end, several research works (e.g., [24, 25, 26, 27]) 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 [24], ICStore [25] and SPANStore [26] and SCFS [27] 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 [25] and SPANStore [26]. Other multi-cloud systems that do address malice in systems (e.g., DepSky [24] and SCFS [27]) 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.

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

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

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

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

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

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

- Tolerating 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., [24, 27]).
- 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 [24]. 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 [28]. 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) [29] of reads and writes even in presence of eventually consistent public clouds [23]. 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 [30]) concurrency control that further boosts the scalability of our system with respect to lock-based systems such as SPANStore [26], DepSky [24] and SCFS [27].

Finally, Hybris optionally supports caching of data stored at public clouds, as well as symmetric-key encryption for data confidentiality leveraging trusted Hybris metadata to store and share cryptographic keys. While different caching solutions can be applied to Hybris, we chose to interface Hybris with Memcached[5] distributed cache, with Memcached deployed on the same machines that run ZooKeeper servers.

We implemented Hybris in Java[6] and evaluated it using both microbenchmarks and the YCSB [31] 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. In Section 5.3.3 we present Hybris performance evaluation.

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 reuses Memcached[7], an open source distributed caching system.

Finally, at the heart of the system is the Hybris client, whose library is responsible for interactions with public clouds, RMDS and the caching service. Hybris clients are also responsible for encrypting and decrypting data in case data confidentiality is enabled — in this case, clients leverage RMDS for sharing encryption keys (see Sec. 4.3.7).

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

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

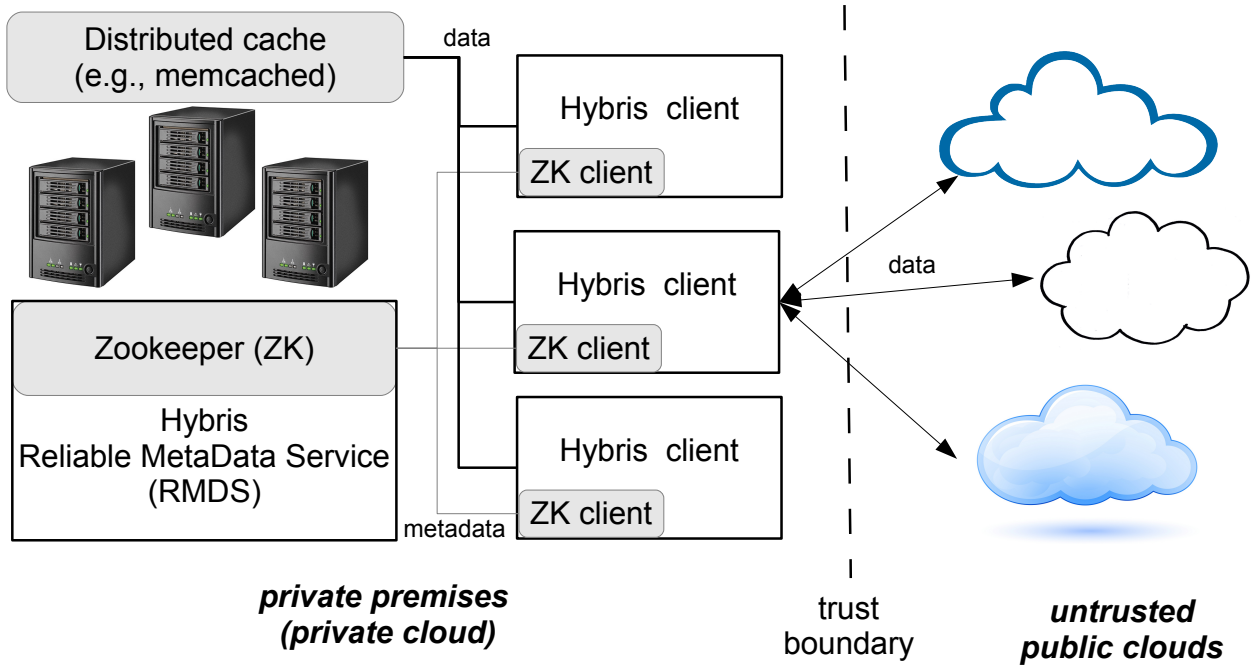


Figure 1: Hybris architecture. Reused (open-source) components are depicted in grey.

System model. We assume an unreliable distributed system where any of the components might fail. In particular, we consider dual fault model, where: (i) the processes on private premises (i.e., in the private cloud) can fail by crashing, and (ii) we model public clouds as potentially malicious (i.e., arbitrary-fault prone [32]) 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).[8] Namely, we assume that the communication among processes located in the private portion of the cloud is partially synchronous [33] (i.e., with arbitrary but finite periods of asynchrony), whereas the communication among clients and public clouds is entirely asynchronous (i.e., does not rely on any timing assumption) yet reliable, with messages between correct clients and clouds being eventually delivered.

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

Finally, for simplicity, we assume an adversary that can coordinate malicious processes

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

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

Hybris data model and semantics. Similarly to commodity public cloud storage services, Hybris exports a key-value store (KVS) API; in particular, Hybris address space consists of flat containers, each holding multiple keys. The KVS API features four main operations: (i) PUT$(cont, key, value)$, to put $value$ under key in container $cont$; (ii) GET$(cont, key, value)$, to retrieve the value; 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 [29]) 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 [30]. 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 its RMDS component which maintains metadata associated with each key-value pair. In the vein of Farsite [34], 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. 4.3.5 for details) is in the way it combines timestamp-based lock-free multi-writer concurrency control within RMDS with garbage collection (Sec. 4.3.4) 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

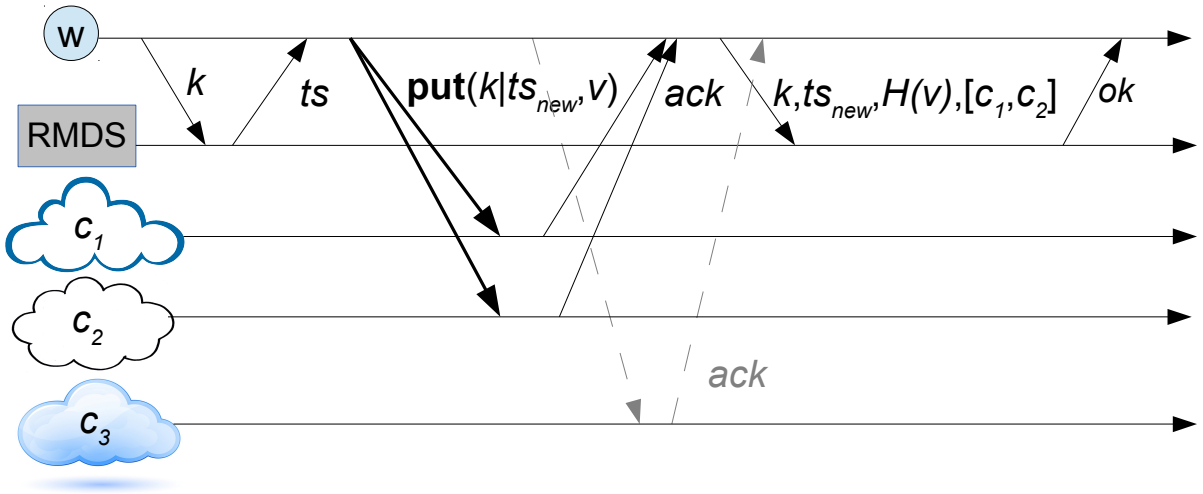


Figure 2: Hybris PUT protocol illustration ($f = 1$). Common-case communication is depicted in solid lines.

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

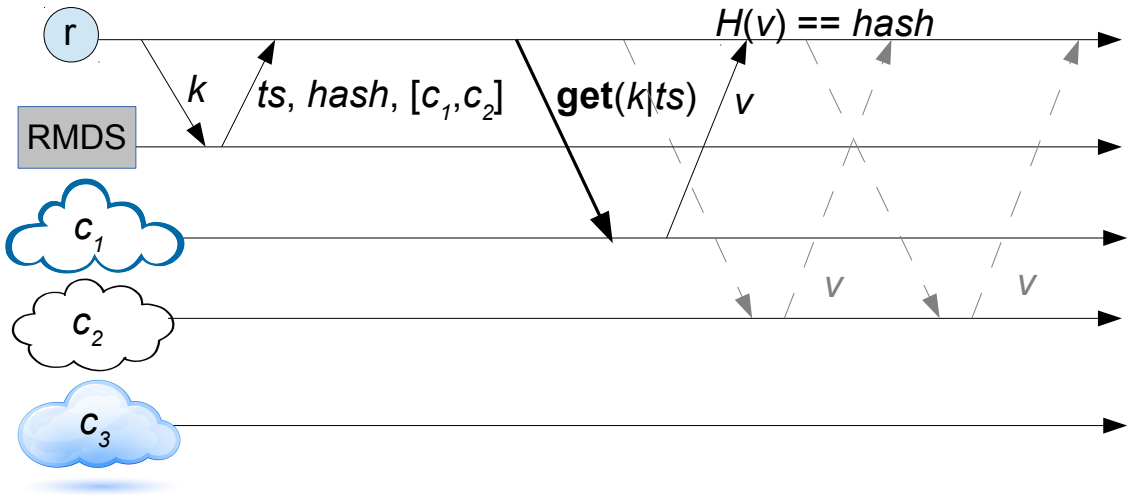


Figure 3: Hybris GET protocol illustration ($f = 1$). Common-case communication is depicted in solid lines.

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*.[9]

In specific corner cases, caused by concurrent garbage collection (described in Sec. 4.3.4), failures, repeated timeouts (asynchrony), or clouds' inconsistency, the client has to take additional actions in GET (described in Sec. 4.3.5).

4.3.4 Garbage Collection

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

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

4.3.5 GET in the worst-case

In the context of cloud storage, there are known issues with weak, e.g., eventual [23] consistency. With eventual consistency, even a correct, non-malicious cloud might deviate from atomic semantics (strong consistency) and return an unexpected value, typically a stale one. In this case, sequential common-case reading from $f + 1$ clouds as described in Section 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. 4.3.4).

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. 4.3.4), a client needs to make sure it always compares the values received from clouds to the most recent key metadata. This can be achieved in two ways: (i) by simply looping the entire GET including metadata retrieval from RMDS, or (ii) by looping only **get** operations at $f + 1$ clouds while fetching metadata from RMDS only when metadata actually changes.

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

[9] 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.

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[10].
   (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 [35] 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. 4.3.4) might systematically and indefinitely often remove the written values before the client manages to retrieve them.[11]

4.3.6 DELETE and LIST

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

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

4.3.7 Confidentiality

Adding confidentiality to Hybris is straightforward.To this end, during a PUT, just before uploading data to $f+1$ public clouds, the client encrypts the data with a symmetric cryptographic key k_{enc}. Then, in the final step of the PUT protocol (see Sec. 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

[10] For simplicity, we model the absence of a value as a special NULL value that can be hashed.
[11] 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.

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}.

4.3.8 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 $k \geq f + 1$.

4.4 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[12] into a common lightweight interface that masks the small differences across native client libraries.

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

4.4.1 ZooKeeper-based RMDS

We layered Hybris implementation over Apache ZooKeeper [28]. In particular, we durably store Hybris metadata as ZooKeeper znodes; in ZooKeeper znodes are data objects addressed

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

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.[13]

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 [30] 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. 4.3.5) we use the concept of ZooKeeper watches (set by e.g., **getData**) which are subscriptions on znode update notifications. We use these notifications in Step 4 of the algorithm described in Section 4.3.5.

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

[13] 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.

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 [36], 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.

Full Hybris evaluation can be found in our 2014 Symposium on Cloud Computing (SoCC) paper [37].

5 Adaptive Edge Platform: *Using BitTorrent to Reduce Bandwidth Cost*

In this world of the internet, people are increasingly opting for cloud storage services, referred to as Personal clouds. Such services are attracting many customers for the wide range of services they are offering, including, but not limited to, accessibility, sharing and syncing. All these advantages in addition to others, make these services very attractive for end- users. This has resulted in an ever increasing number of users of these services. This means that a lot of money should be provisioned for storage and bandwidth in order to cope with this increasing demand.

When a client adds a new file to his personal folder, the content is divided into small entities called “chunks" that are pushed to the storage servers while the meta-data is kept in a different server. Once the file is uploaded, all the synchronized devices will be notified of the addition of the new file and will request a copy of it from the cloud storage servers. This will result in the repeated distribution of the same content in a short period of time. From a technical point of view, most of them use HTTP as a transfer protocol and miss the opportunity to benefit from the common interest of users in the same content. In such scenarios, the cloud can benefit from the clients’ upload capacities to save its bandwidth. In this context, we propose to introduce the BitTorrent [38] protocol when the load on a specific file becomes high. In fact, the efficiency of the peer-assisted paradigm makes it especially suitable for files shared between a set of devices.

Unfortunately, the use of BitTorrent may incur a longer download time compared to HTTP especially for small files. According to previous studies, these small files form a very high percentage of the data stored in personal cloud servers; 99% of the files are of size smaller than 16 MB according to [39]. The main challenge is to decide when it is worth switching to BitTorrent. The key elements in making the decision are the gain in download time and the peers’ contribution. The former represents the difference in download time between HTTP and BitTorrent. The latter measures the total amount of data that can be obtained from the peers. To our knowledge, there were no previous studies that compare the BitTorrent and HTTP protocols for distributing small files. Thus, it comes the need to draw a complete comparative study between both protocols.

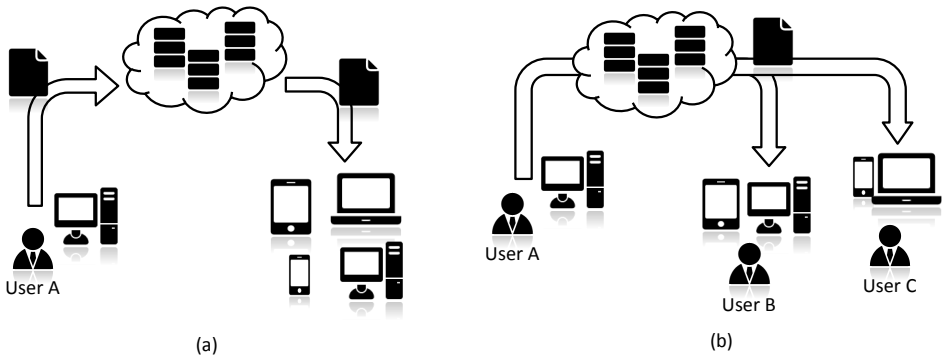


Figure 4: Synchronisation and sharing in personal cloud systems

Let’s consider a classic personal cloud system where the storage server is responsible

for storing the clients files and managing the corresponding requests. Each client i has an upload bandwidth u_i and a download bandwidth d_i. 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 downloaded by all the other synchronized devices of the user. (Part (a) of Fig. 4)
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. (Part (b) of Fig. 4)

Both cases can be modeled by the problem of distributing a file f of size F from the cloud server to L distinct nodes. We denote by $\mathcal{S}$ the set of cloud seeds serving f and by $u(\mathcal{S})$ their aggregated upload speed dedicated to f. In personal clouds, file synchronization and content distribution follow a client-server model centralized in the cloud storage provider. As mentioned above, the download protocol adopted by these providers is HTTP. The choice of this protocol is made because it uses the port 80 which is generally kept open. While this kind of architecture (client-server) is appropriate for some use cases, it is not optimal when the number of nodes requesting the same content is high, as it might result in bandwidth bottlenecks in the cloud.

A possible solution is to benefit from the high number of requests and use the clients' spare upload bandwidth to offload the server. The main idea is to switch from HTTP to BitTorrent upon detection of an increasing number of requests on a specific content. The architecture of our system consists of a Cloud storage system and a personal cloud client. The main components of the resulting architecture are:

- ***Cloud side:***
  - *OpenStack Swift*: The client's files are replicated in each of our *Storage nodes* in order to keep the system in a consistent state in the face of drive failures. The *Proxy Server* is the entity responsible for handling the requests. For each request, it looks for the location of the requested object and routes the request accordingly. This server is also responsible for monitoring the incoming requests and upon detection of a certain mass on a specific content, decides to switch to BitTorrent.
  - *Torrent server*: This server is triggered when the Proxy decides to switch to the BitTorrent protocol for a specific content. Then, it runs a seed whose role is to extract the requested chunks from the storage nodes. The server will generate later a corresponding meta-info (*.torrent*) file that will be transmitted to the client.
- ***Personal Cloud Client side:*** The personal cloud client is extended with an implementation of a BitTorrent client to allow a transparent switch to BitTorrent.

Our system is able to transparently switch from a regular client-server model using the HTTP to a peer-to-peer model using the BitTorrent protocol, following a protocol decision algorithm (see section 5.2). During content download, users do not need to take any action for the switching of protocols. The specific time at which the switch is made is configurable depending on the input of the algorithm.

In this context, we foresee two challenges that we will address throughout this document. The first challenge is 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?*

The second challenge is how to allocate the limited bandwidth on the data center side to a different set of swarms to maximize content download throughput to also increase the efficiency and responsiveness of the system compared to current solutions. For this, we propose a smart seeding strategy which grants bandwidth according to swarm characteristics.

For the rest of this document, we will consider the following swarm-level notation:

Parameter	Description
F	size of the requested file f
$\mathcal{S}$	set of providers of f (seeder nodes)
$\mathcal{L}$	set of requesters of f (leecher nodes), L= $\lvert\mathcal{L}\rvert$ is the number of requesters
$\mathcal{I}$	set of all the nodes, $\mathcal{I} = \mathcal{L} \cup \mathcal{S}$
u_i	upload speed of node $i \in \mathcal{L}$
d_i	download speed of node $i \in \mathcal{L}$
d_{min}	$d_{min} = \min_{i \in \mathcal{L}}(d_i)$ download speed of the slowest leecher requesting f
$u(\mathcal{A})$	$u(\mathcal{A}) = \sum_{i \in \mathcal{A}} u_i$ aggregated upload bandwidth of $\mathcal{A} \subseteq \mathcal{I}$
$d(\mathcal{A})$	$d(\mathcal{A}) = \sum_{i \in \mathcal{A}} u_i$ aggregated download bandwidth of $\mathcal{A} \subseteq \mathcal{I}$
C_f	$C_f = \{(u_i, d_i),\ \forall i \in \mathcal{L}\}$ set of upload and download bandwidths of all the leechers interested in f.

5.1 Client-server Versus Peer-Assisted File Distribution

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 [40] 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. We ran several experiments distributing files of sizes 1, 5 and 10 MB using a unique seed. We used the following common ADSL bandwidth settings: the clients had an upload bandwidth u_i=1 Mbps, a download bandwidth d_i=2 Mbps and the bandwidth allocated by the cloud to each exchanged file was $u(\mathcal{S})$=5 Mbps. We measured the average download time in HTTP and BitTorrent for each experiment and calculated the corresponding gain or loss in download time. We also measured the total amount of data

contributed by the peers. We report the results in table 5.1a. All the download times in the table are in seconds.

Clients count	1 MB file				5 MB file			
	HTTP	BT	Time diff	Data from peers	HTTP	BT	Time diff	Data from peers
2	4 s	5.51 s	-1.51 s	236.13 KB	20 s	21.52 s	-1.52 s	2.9 MB
3	4.8 s	5.47 s	-0.67 s	819.6 KB	24 s	21.69 s	+2.31 s	6.02 MB
4	6.4 s	6.03 s	+0.37 s	1.57 MB	32 s	23.06 s	+8.94 s	7.84 MB
5	8 s	6.25 s	+1.75 s	1.64 MB	40 s	24.05 s	+15.95 s	11.59 MB

Table 5.1a: Measured download times for small files using HTTP and BitTorrent. 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.

We notice that, with four clients downloading a 1 MB file, BitTorrent can reduce the download time compared to HTTP. The peers contribution can reach 40% of the total data volume in some cases.

In this section, we present our estimation for the distribution time of small files via BitTorrent. We also introduce the gain and offload ratios in order to measure the trade-off between this protocol and HTTP.

5.1.1 The Distribution Time for Small Files in BitTorrent

Background To get an estimation of the download time in BitTorrent-like systems, we borrow the following formula proposed in [41] by Kumar and Ross:

$$T_{pa}\left(u(\mathcal{S}), C_f, F\right) = \frac{F}{\min\left\{d_{min}, \frac{u(\mathcal{I})}{\mathrm{L}}, u(\mathcal{S})\right\}}, \tag{1}$$

where T_{pa} is the minimum time needed to distribute a file of size F to L leechers. This time depends on the download speed of the slowest peer d_{min}, the aggregated upload bandwidth of all the nodes divided equally between all the L 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: $d_{min} \leq \min\left\{\frac{u(\mathcal{I})}{\mathrm{L}}, u(\mathcal{S})\right\}$ and $d_{min} \leq \frac{u(\mathcal{L})}{\mathrm{L}-1}$:
   In this case, the download speed of the peers is limited by the download bandwidth of the slowest peer in the swarm d_{min}.

2. Case B: $d_{min} \leq \min\left\{\frac{u(\mathcal{I})}{\mathrm{L}}, u(\mathcal{S})\right\}$ and $\frac{u(\mathcal{L})}{\mathrm{L}-1} \leq d_{min}$:
   In Case B, the transfer is limited by the maximum speed at which a leecher can get data from the other leechers, that is $\frac{u(\mathcal{L})}{\mathrm{L}-1}$.

3. Case C: $\frac{u(\mathcal{I})}{\mathrm{L}} \leq \min\{d_{min}, u(\mathcal{S})\}$:
   The transfer bottleneck in this case is limited by the aggregated upload speed of the network $u(\mathcal{I})$ divided equally between the L leechers.

4. Case D: $u(\mathcal{S}) \leq \min\left\{d_{min}, \frac{u(\mathcal{I})}{\mathrm{L}}\right\}$:
   In this case, the upload bandwidth of the seed $u(\mathcal{S})$ is the maximum limit at which each peer can download "fresh" content.

For each of the cases listed above, the authors in [41] 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 $(\mathrm{L}-1)$ leechers at a rate $x_i(t)$, where $x_i(t) \leq s_i(t)$, as shown in Figure 5. 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 $\mathrm{L}-1$ other leechers.

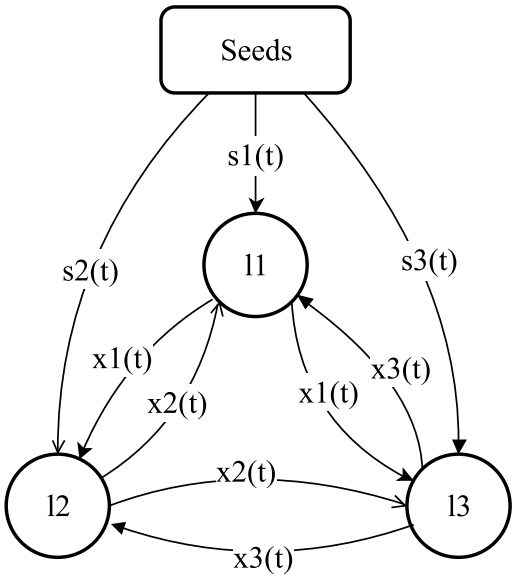


Figure 5: General distribution scheme structure: Leecher li ($i \in \{1,2,3\}$) downloads "fresh" data at the rate $s_i(t)$ from the seeds. The data is replicated later to the other 2 leechers at a rate $x_i(t) < s_i(t)$.

To 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, the upload speed of the seeds $u(\mathcal{S})$, and the set of upload and download speeds of all the leechers C_f. For a complete proof and more details regarding these formulas, we kindly refer the reader

to the original paper [41].

$$s_i(t) = \begin{cases} \frac{u_i \times d_{min}}{u(\mathcal{L})} & \underline{Case\ A} \\ \frac{u_i - u(\mathcal{L})}{\mathrm{L}-1} + d_{min} & \underline{Case\ B} \\ \frac{u_i - u(\mathcal{L})}{\mathrm{L}-1} + \frac{u(\mathcal{I})}{\mathrm{L}} & \underline{Case\ C} \\ \frac{u_i \times u(\mathcal{S})}{u(\mathcal{L})} & \underline{Case\ D} \end{cases} \quad (2)$$

Adding the BitTorrent overheads One of the limitations of (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 considered swarms whose size ranged from 2 to 5. We considered the same bandwidth settings as in the experiments used in Table 5.1a. 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 5.1b, where the estimated and experimental download times, and the absolute error are all measured in seconds.

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%

Table 5.1b: Estimated versus experimental distribution time with BitTorrent of a 1MB file.

As we can see in Table 5.1b, 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 *.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. For more details about the experimental evaluation of α_{bt}, please refer to Fig. 6.

- *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 [42], 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: [14]
  $$\eta = 1 - \sum_{n_i=0}^{N-1} \frac{1}{N} \left(\frac{N - n_i}{N\,(n_i + 1)} \right)^k,$$
  where N is the number of pieces of the served file and k the number connections a peer has.
  The authors in [42] 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 $\eta = 0.7069$, 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 an accurate estimation of the download time in BitTorrent as follows:

$$T_{bt}\left(u(\mathcal{S}), C_f, F\right) = \frac{F}{\min\left\{d_{min}, \frac{u'(\mathcal{I})}{\mathrm{L}}, u(\mathcal{S})\right\}} + \alpha_{bt}, \tag{3}$$

where $u'(\mathcal{I}) = u(\mathcal{S}) + \eta\ u(\mathcal{L})$ is the scaled aggregated upload speed of all the nodes, including both the seeders and the leechers.

Validation of α_{bt} and T_{bt} To validate our extended formulas of the download time, we run repeated experiments using a 1 MB file. The experimental scenario is to distribute the file via BitTorrent starting with a unique seed. The reasons behind the choice of such a small file lies in the fact that in personal cloud systems most of the files are in the order of a few megabytes in size. The experimental setting is the following: the upload bandwidth of the cloud dedicated for the file is $u(\mathcal{S}) = 5$ Mbps. The number of clients ranges from 2 to 10. Each of them has an upload bandwidth of 1 Mbps and a download bandwidth of 2 Mbps.

We repeated each experiment 5 times and measured the average values of the download time and the overhead α_{bt} for each client. Figure 6 represents a box-plot of the time interval between the moments when the clients are launched and when they start downloading the file. This time interval corresponds to α_{bt}. We notice that the average value of the overhead is about 2.5 seconds for our architecture. Using this value for the discovery overhead

[14] The rectified version of [42] which contains the correct expression of η can be found at: `http://users.encs.concordia.ca/~dongyu/paper/bittorrent.pdf`

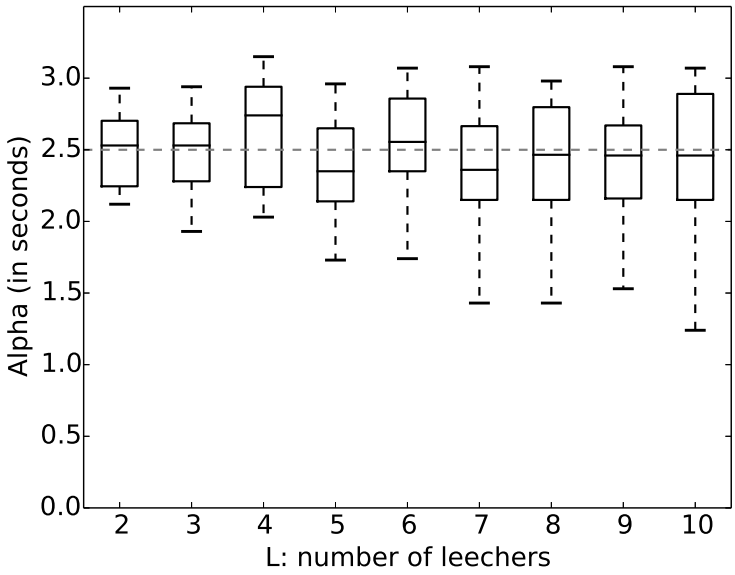


Figure 6: The overhead α_{bt}: time before clients start downloading for a file of 1 MB size

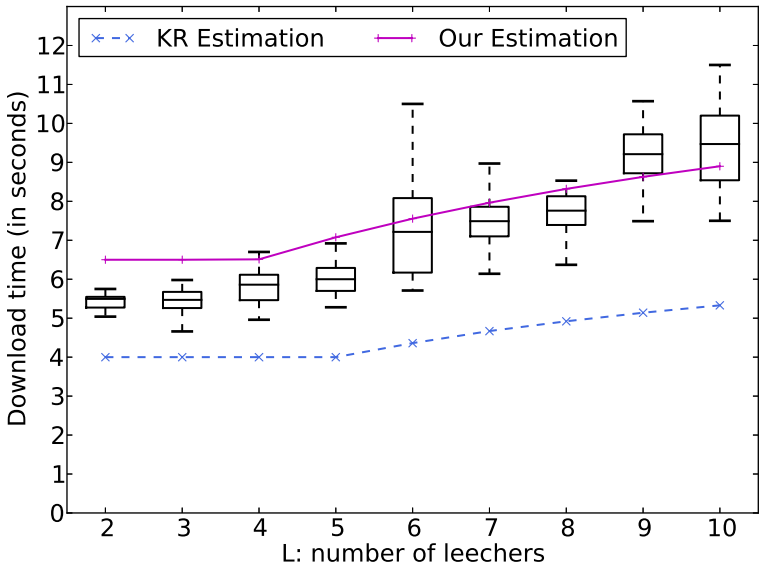


Figure 7: Comparison of the experimental download time (boxplot), our estimation and the estimation proposed in [41] (KR Estimation)

($\alpha_{bt} = 2.5$), Figure 7 compares our estimation, the one proposed in [41] and the experimental results. We can clearly see that the error can reach 40% in the case of Kumar and Ross's estimation. This error is reduced to about 10% using the estimation we propose.

5.1.2 Comparative Analysis of BitTorrent Relative to HTTP

We introduce here two metrics that measure the comparative efficiency of HTTP and BitTorrent, especially for small files. These metrics are the gain and offload ratios. The former

represents the normalized ratio of the difference between download times. The latter models the amount of data offloaded from the cloud.

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

$$Gain(u(\mathcal{S}), C_f, F) = \frac{T_{cs}(u(\mathcal{S}), C_f, F) - T_{bt}(u(\mathcal{S}), C_f, F)}{T_{cs}(u(\mathcal{S}), C_f, F)},$$

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} or the bandwidth of all the seeds $u(\mathcal{S})$ divided equally between the L clients. T_{cs} can be simply defined as follows:

$$T_{cs}\left(u(\mathcal{S}), C_f, F\right) = \frac{F}{\min\left\{d_{min}, \frac{u(\mathcal{S})}{\mathrm{L}}\right\}}. \tag{4}$$

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 $\min\left\{d_{min}, \frac{u'(\mathcal{I})}{\mathrm{L}}, u(\mathcal{S})\right\}$ and $\min\left\{d_{min}, \frac{u(\mathcal{S})}{\mathrm{L}}\right\}$:

1. Case I: $d_{min} \leq \frac{u(\mathcal{S})}{\mathrm{L}}$ and $d_{min} \leq \min\left\{\frac{u'(\mathcal{I})}{\mathrm{L}}, u(\mathcal{S})\right\}$:
   In this case, the bottleneck in HTTP and BitTorrent is the download speed of the slowest peer. The corresponding download times are: $T_{cs} = \frac{F}{d_{min}}$ and $T_{bt} = \frac{F}{d_{min}} + \alpha_{bt}$.

2. Case II: $\frac{u(\mathcal{S})}{\mathrm{L}} \leq d_{min}$ and $d_{min} \leq \min\left\{\frac{u'(\mathcal{I})}{\mathrm{L}}, u(\mathcal{S})\right\}$:
   In Case II, the bottleneck in HTTP is $\frac{u(\mathcal{S})}{\mathrm{L}}$, while it is equal to d_{min} in BitTorrent. The corresponding download times are: $T_{cs} = \frac{F \times \mathrm{L}}{u(\mathcal{S})}$ and $T_{bt} = \frac{F}{d_{min}} + \alpha_{bt}$.

3. Case III: $\frac{u'(\mathcal{I})}{\mathrm{L}} \leq \min\{d_{min}, u(\mathcal{S})\}$:
   In this case, the bottleneck in BitTorrent is $\frac{u'(\mathcal{I})}{\mathrm{L}}$. And since $u(\mathcal{S}) \leq u'(\mathcal{I})$ and $\frac{u'(\mathcal{I})}{\mathrm{L}} \leq d_{min}$, this means that $\frac{u(\mathcal{S})}{\mathrm{L}}$ is always $\leq d_{min}$. Thus, in this case, $T_{cs} = \frac{F \times \mathrm{L}}{u(\mathcal{S})}$ and $T_{bt} = \frac{F \times \mathrm{L}}{u'(\mathcal{I})} + \alpha_{bt}$.

4. Case IV: $u(\mathcal{S}) \leq \min\left\{d_{min}, \frac{u'(\mathcal{I})}{\mathrm{L}}\right\}$:
   Since $\frac{u(\mathcal{S})}{\mathrm{L}} \leq u(\mathcal{S})$ and $u(\mathcal{S}) \leq d_{min}$, this means that $\frac{u(\mathcal{S})}{\mathrm{L}}$ is always $\leq d_{min}$. In this case, $T_{cs} = \frac{F \times \mathrm{L}}{u(\mathcal{S})}$ and $T_{bt} = \frac{F}{u(\mathcal{S})} + \alpha_{bt}$.

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

$$Gain\left(u(\mathcal{S}), C_f, F\right) = \begin{cases} -\frac{\alpha_{bt} \times d_{min}}{F} & \underline{Case\ I} \\ 1 - \frac{u(\mathcal{S})}{\mathrm{L}.d_{min}} - \frac{\alpha_{bt}.u(\mathcal{S})}{F \times \mathrm{L}} & \underline{Case\ II} \\ 1 - \frac{u(\mathcal{S})}{u'(\mathcal{I})} - \frac{\alpha_{bt}.u(\mathcal{S})}{F \times \mathrm{L}} & \underline{Case\ III} \\ 1 - \frac{1}{L} - \frac{\alpha_{bt} \times u(\mathcal{S})}{F \times \mathrm{L}} & \underline{Case\ IV} \end{cases} \tag{5}$$

The 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:

$$\begin{aligned} Offload\left(u(\mathcal{S}), C_f, F\right) &= \frac{data\ from\ peers}{total\ data\ sent} = 1 - \frac{data\ from\ cloud}{total\ data\ sent} \\ &= 1 - \frac{\sum\limits_{i \in \mathcal{L}} \int_0^{T_{bt}\left(u(\mathcal{S}), C_f, F\right)} s_i(t)dt}{F \times \mathrm{L}} \end{aligned}$$

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:

$$Offload\left(u(\mathcal{S}), C_f, F\right) = \begin{cases} 1 - \frac{1}{\mathrm{L}} & \underline{Case\ A} \\ \frac{\eta.u(\mathcal{L})}{\mathrm{L} \times d_{min}} & \underline{Case\ B} \\ 1 - \frac{u(\mathcal{S})}{u'(\mathcal{I})} & \underline{Case\ C} \\ 1 - \frac{1}{\mathrm{L}} & \underline{Case\ D} \end{cases} \tag{6}$$

Validation of the Gain and Offload Ratios Since the gain is a key parameter in the protocol decision algorithm, it is important to verify the accuracy of our estimation compared to real experimental values. We ran experiments using the same bandwidth distribution as mentioned above. The goal is to compare the experimental and the estimated gain ratios when distributing a file to a set of nodes whose size range from 2 to 12 nodes. The file size varies from 1 to 25 MB. Figure 8 represents a 3-dimensional plot of the results. The *Gain*$= 0$ plane represents the threshold $\tau = 0$. As we can see , the experimental and estimated surfaces are very close and the difference between them is slight.

Figures 9a, 9b and 9c present some vertical sections of the previous plot for files of size 1, 5 and 25 megabytes along with the corresponding estimation and experimental values of the offload ratio. We notice that the estimations are very close to the experimental results in most cases. For instance, for the smallest file of size 1 MB, the error in the gain estimation is moderate for very small swarms with only 2 or 3 clients. That error could represent an increase in the download time of a few seconds. However, we notice that the bigger the swarms is, the closer the estimation gets, in a way that the error becomes negligible for swarms of size ≥ 4 clients. For bigger files, the estimation is very accurate and the error does not exceed 5% in most cases.

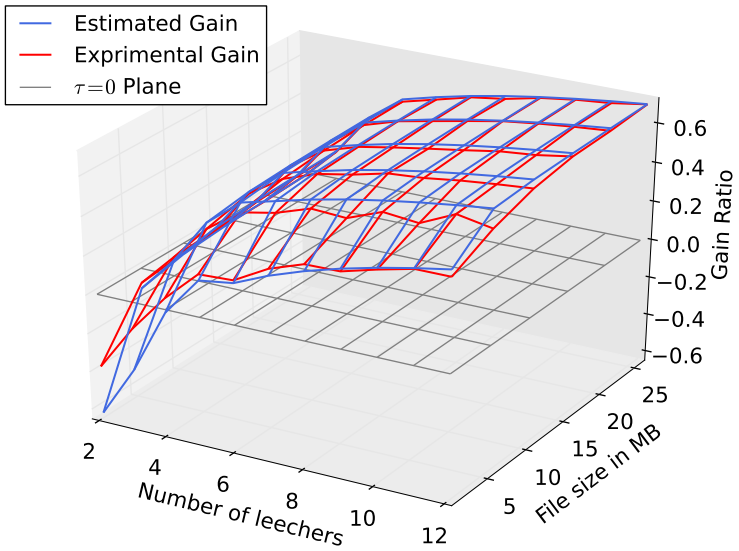
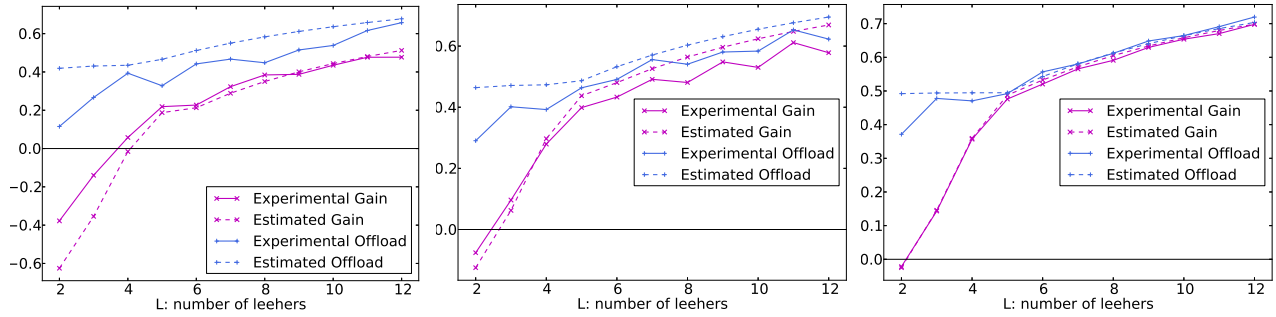


Figure 8: Experimental versus estimated gain ratios and the $\tau = 0$ gain plane



(a) 1 megabyte file (b) 5 megabyte file (c) 25 megabyte file

Figure 9: Estimated versus experimental gain and offload ratios for files with different sizes

5.2 Switching Algorithm

In this section, we first study the criteria that can be considered in the definition of the switching point. We present later our proposed algorithm for the management of the download protocols.

5.2.1 Switching Criteria

We presented in the previous section two key parameters that can help us measure the trade-off 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$\geq$ 2. In this case, the overall offload ratio will be the highest possible. However, 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[15], it is possible to loosen that constraint, and tolerate delays of up to 50% (which corresponds to $\tau = -0.5$), for instance. To get an idea about the tradeoff between HTTP and BitTorrent, please refer to Table 5.2a in which we measured the overall offload ratio based on different values of τ.

5.2.2 Implementation of the Switching Algorithm

For the management of the download protocols, we propose Algorithm 1, which is executed upon the arrival of each new download request on a certain file.

[15] 96% of Dropbox clients use the free version of the service (Souce: `http://www.economist.com/blogs/babbage/2012/12/dropbox`)

We suppose that our system keeps track of the state of each file f as a boolean value $switched_f$, where $switched_f$=*true* if the current download protocol is BitTorrent (the switch has already taken place) and *false* otherwise.

Algorithm 1 Protocol Decision Algorithm

Require: τ: the gain constraint
Require: $switched_f$: the state of file f
Require: F: the size of file f
Require: $u(\mathcal{S})$: the upload speed of the seeder nodes
Require: $C_f = \{(u_i, d_i),\ \forall i \in \mathcal{L}\}$: set of upload and download bandwidths of all the leechers interested in f.

```
if ( not switched_f ) then
  calculate Gain(u(S), C_f, F)
  if (Gain(u(S), C_f, F) ≥ τ ) then
    create a .torrent
    launch a BT seed in the cloud
    for all clients requesting f do
      get the .torrent from the server
      launch a BT leecher
      start BT transfer
    end for
    switched_f=true
  else
    download the file via HTTP
  end if
else
  send the .torrent to the new requester
  launch a BT leecher inside that requester
end if
```

The algorithm works as follows: Whenever there is a new download request on a file f, the system verifies the download protocol already in use to distribute f. If it is being downloaded by the default protocol (HTTP), the system computes the estimated gain and compares it with τ. If the resulting gain is below the constraint, the file will be sent to the requester via HTTP. Otherwise, the distribution protocol will be switched to BitTorrent. A *.torrent* file will be created and sent to all the clients requesting f. 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 of these clients. After this phase, the clients will start downloading the file in BitTorrent, while offloading the cloud from doing all the serving.

5.2.3 Validation of the Algorithm

To validate our proposal and measure how much bandwidth can be saved using our algorithm, we used a real trace of UB1 [43]. The trace was collected based on the behavior of real users, each represented by a hash code for privacy reasons. Each line of the trace represents an operation of download or upload performed by a user on a file. For each operation, several information were collected, including: the time-stamp, the type of operation ('up' or

'down'), the hash and size of the file in question and finally the identifier of the user. For a period of 30 hours, 3,318,950 operations on 1,887,247 distinct files were logged including 2,231,791 upload operations (67.24%) and 1,087,159 (32.76%) download operations. The total downloaded volume was about 1,240.25 GB. The total number of different users involved in this trace was 19,319.

We applied our algorithm on the trace using the following settings:

- The upload speed of the seed: $u(\mathcal{S}) = 2$ Mbps. We remind our reader that $u(\mathcal{S})$ 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.
- The clients are homogeneous and have an upload and download speed of 512 Kbps and 1 Mbps, respectively.
- The peers discovery overhead is $\alpha_{bt} = 2.5$ seconds.

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).

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

Table 5.2a 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.

Monetary Cost 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 deliverable, the standard charging rates for data transfer were[16]:

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

Effect of file bundling Bundling consists in grouping a batch of small files that need to be transferred as a single object. This technique is used by Dropbox [44] in an attempt to reduce both transmission latency and control overhead.

If we take a look at equation (5), we notice that the gain ratio and the file size F are related in a way that if F increases, the gain will increase too. Similarly, file bundling should presumably increase the overall offload too. Here, we study the effect of applying this technique in our trace. For a given "bundling period", we group the files that are requested by the same users and consider them as a single file, so that a single *.torrent* file is created for all of them.

Bundling Period	Constraint	Offloaded Volume	Overall Offload%
10 seconds	$\tau = -0.2$	213.97 GB	17.2526%
	$\tau = 0.0$	140.95 GB	11.3658 %
30 seconds	$\tau = -0.2$	214.43 GB	17.2895 %
	$\tau = 0.0$	140.95 GB	11.3658 %

Table 5.2b: Results using file bundling

Table 5.2b shows the results of grouping files considering 2 different bundling periods. We notice that, compared to the previous results, bundling is not very effective in this scenario: a slight improvement of the overall offload percentage in the order of 0.55% for $\tau = -0.2$ and about 0.26% for $\tau = 0$. Even with a long grouping period of 30 seconds, the increase of the overall offload percentage remains limited: in the order of 0.03% compared to a bundling period of 10 seconds. However, these results do not imply that the use of this technique could not be effective in increasing the offload rate in other systems.

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

5.3 Smart Cloud Seeding for BitTorrent swarms

Following the previous model of operation, the data center could be serving different popular content to different swarms at the same time. Thus, considering the benefits of transparently integrating BitTorrent in the data center to deliver content, we now face the problem of *allocating the limited bandwidth of the data center in a multi-swarm scenario*. Our main motivation is to increase the overall throughput of content delivery to provide a higher number of users with a responsive system which minimizes content download time.

Our goal is to determine the upload bandwidth w_s that the data center should allocate to a swarm $s \in S$, assuming an upper bound on bandwidth consumption $W = \sum_{s \in S} w_s$, in order to maximize the aggregate download bandwidth considering all swarms $\sum_{s \in S} D_s$, where D_s is the aggregate download bandwidth of the peers in swarm s. By maximizing the download speed of swarms, we ensure that download times are minimized and that content is distributed as fast as possible within a restricted bandwidth budget.

5.3.1 Bandwidth Response Model

Our smart seed strategy uses information about the dynamic state of the swarm to properly decide the amount of bandwidth that needs to be allocated. This key piece of information is the *response curve* which represents the swarm aggregate bandwidth as a function of the allocated seeder bandwidth [45]. This function $f_s(w_s)$ embodies information about the current sustainable download bandwidth of the swarm ($a = \sum_{p \in s} a_p$) and the aggregate upload ($u = \sum_{p \in s} u_p$) and download ($d = \sum_{p \in s} d_p$) saturation bandwidth of the swarm (see Fig. 10). These response curves depend on a number of factors like the number of leechers (n) and seeders, their actual bandwidth contribution to the swarm and the current distribution of unique blocks, elements that are naturally dynamic in BitTorrent swarms.

However, bandwidth response curves have a characteristic shape which can be modeled approximately using known information about the state of a swarm. Specifically, we used a family of hyperbolic functions of the form $f_s(w_s) = a + \frac{(d-a)w_s}{w_s+c}$ where w_s is the seeder bandwidth allocated to swarm s, and $c = \frac{d-u}{n}$ is the parameter that shapes the increment rate of the function – the higher the value, the flatter the curve. This parameter is obtained by substituting the coordinate $(x, y) = (\frac{u-a}{n}, u)$ on the bandwidth response curve, and represents the point at which the data center provides enough bandwidth to saturate the upstream links of peers.

The intuition behind this model is initially sketched in [45] and is as follows. When the data center bandwidth allocated to a swarm is zero, the swarm does not receive any new block from the data center seeder and thus, the swarm's aggregate bandwidth is the current download bandwidth of the swarm sustained by any seeder other than the datacenter, which is responsible to inject new blocks (a). If no other seeder is present, a will drop to zero as soon as no new blocks are present for exchange. For the rest of this section we will assume that $a = 0$ for simplicity.

The second interesting point of our response curve model is when the datacenter bandwidth is equal to the average uplink capacity of peers ($\frac{u}{n}$). At this seeder bandwidth, the datacenter is able to inject new blocks to the swarm at enough rate to saturate the uplink

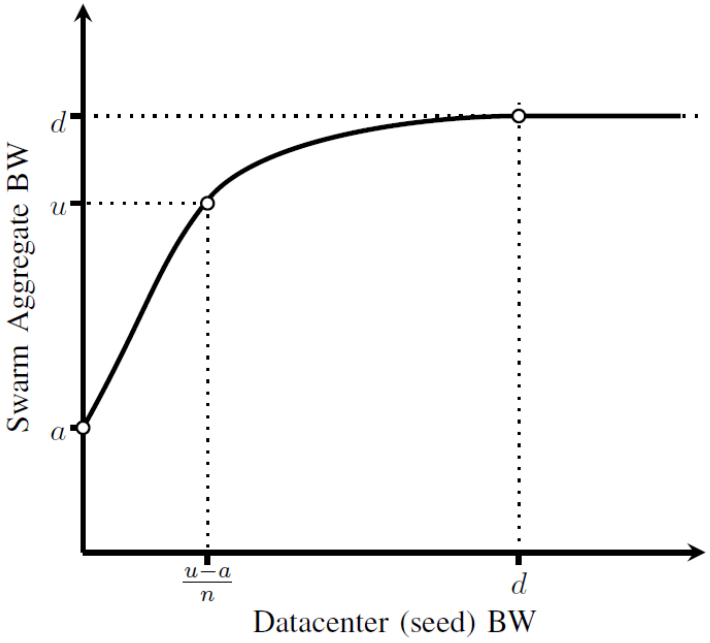


Figure 10: Response of the swarm aggregate bandwidth to the data center allocated bandwidth.

capacity of leechers, achieving an aggregate download speed equal to the aggregate upload speed of leechers.

From this point onwards, since saturated upload links render it unable to redistribute blocks to other peers, any addition to the datacenter bandwidth capacity only benefit the peer receiving this increment, and the curve starts to flatten until the download links of clients saturate. At this point the swarm reaches its maximum aggregate bandwidth and any further bandwidth provisioned to the seed will not have any impact on download times.

Our implementation gathers two different sets of information to build this model: i) static information from our instrumented clients like maximum upload capacity (u_p) and maximum download capacity (d_p) which are measured every time a client upload and download new content directly to and from OpenStack Swift. This values are supposed to rarely change during a client life-time; and ii) dynamic information gathered also from our instrumented client like the effective download speed of a peer belonging to a swarm (a_p) and the number of peers in a swarm (n) obtained from the public information on the tracker. Because of the dynamic nature of swarms, we update this information every 5 minutes in our implementation.

5.3.2 Optimization problem and implementation

The previous function closely resembles the bandwidth response curve of a swarm composed of heterogeneous peers in terms of upstream and downstream bandwidth. Given that our main goal is to maximize the aggregate bandwidth of all swarms, our problem is therefore stated as the following constrained optimization problem.

$$\begin{aligned} \textbf{max} \quad & F(w_1, \ldots, w_m) = \sum_{s \in S} f_s(w_s) \\ \textbf{where} \quad & f_s(w_s) = a_s + \frac{(d_s - a_s) w_s}{w_s + c_s} \\ \textbf{subject to} \quad & \sum_{s \in S} w_s \leq W \\ & w_s \geq 0 \end{aligned} \tag{7}$$

The constraints restricts the solution space to those allocation which yield positive bandwidth allocations and those in which the sum of the allocated bandwidth to each swarm cannot exceed the bandwidth budget of the data center.

An optimal solution $w^* = [w_1^*, \ldots, w_n^*]$ exists for Equation (7) because the objective function is continuously differentiable, strictly increasing and concave. This kind of optimization problems have been studied extensively in the literature ([46][47] as examples) and provide a computationally efficient solution using Lagrange multipliers as shown in Algorithm 2. We omit the details of the proofs and the mathematical development for the sake of clarity, and we refer the reader to [48].

Algorithm 2 Data center Bandwidth Allocation Algorithm

Require: W {Data center bandwidth budget}
Require: $\{(u_1, d_1, a_1, c_1), \ldots, (u_m, d_m, a_m, c_m)\}$ {Swarms' parameters}
Sort increasingly the set of swarms by its marginal value $\frac{d_i - a_i}{c_i}$
Compute largest k such that

$$\frac{\sqrt{c_k(d_k - a_k)}}{\sum_{i=1}^{k} \sqrt{c_i(d_i - a_i)}} (W + \sum_{i=1}^{k} c_i) - c_k \geq 0$$

Set $w_j = 0$ for $j > k$, and for $1 \leq j \leq k$, set:

$$w_j = \frac{\sqrt{c_j(d_j - a_j)}}{\sum_{i=1}^{k} \sqrt{c_i(d_i - a_i)}} (W + \sum_{i=1}^{k} c_i) - c_j$$

return $(x_1, \ldots, x_m)$

The computational complexity of this algorithm is $O(n \log n)$ which is dominated by the initial sorting algorithm. In practice, this algorithm would be computed once any of the parameters that specify the response curve changes which happens only when swarm membership change –e.g. a client leave or join the swarm or when a client finishes downloading the content and become a seeder.

5.3.3 Evaluation

We have successfully integrated transparently BitTorrent within our open source personal cloud storage client which currently uses OpenStack swift as a storage back-end. Besides, our smart seeding strategy has been tested in a real setting using PlanetLab [49] nodes as well as with a simulator to speed up testing [50].

For our simulation experiments, we used standard values for the BitTorrent protocol (64 KB per block, 30s between optimistic unchokes, 10s between regular unchokes) and selected upload and download bandwidths from a distribution used in other works related to BitTorrent [50]. Our simulated setup consists of 300 swarms which membership sizes are drawn from the distribution Zipf($z = 2.4$) which leads to a small number of big swarms and a higher number of small swarms, a typical distribution for the popularity of files. We included in our evaluation three other strategies to distribute bandwidth among swarms in addition to our smart seeding strategy for comparison purposes:

i. Equal sharing, which grants the same up-link capacity to all swarms ($w_i = \frac{W}{m}$)

ii. Proportional sharing, which allocates bandwidth proportionally to the size of each swarm in terms of peers ($w_i = \frac{n_i}{\sum_j^m n_j} * W$)

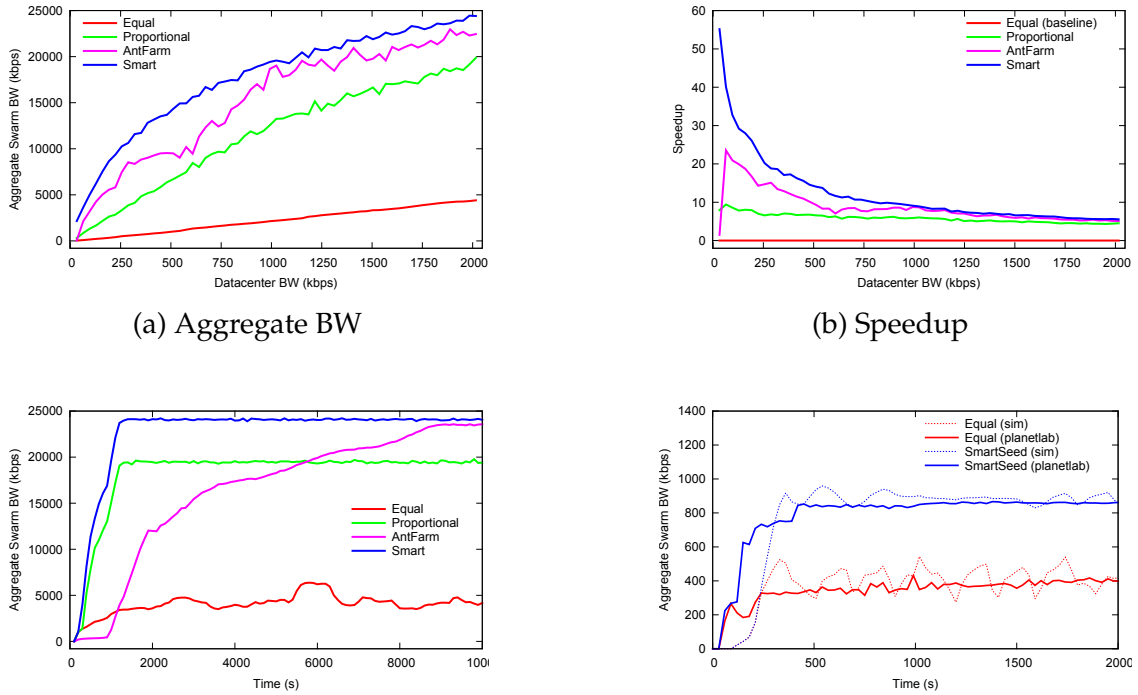
iii. The Antfarm strategy [45], the most similar system in terms of objectives to our mechanism –i.e. maximize aggregate bandwidth.

To give an overview of the Antfarm mechanism, at the beginning it allocates a small amount of seeder bandwidth to every swarm, and then allocates the remaining bandwidth in small increments to swarms with the highest increase since the last update. This way, Antfarm is able to slowly build response curves for each swarm and have an estimate of the swarm performance as a function of seeder bandwidth. This response curve is computed fitting a piecewise-linear function to the set of measurements it takes periodically producing a shape similar to our model (see Fig. 10). In steady state, Antfarm uses a greedy hill-climbing algorithm to allocate bandwidth to swarms with the highest gradient.

Fig. 11a shows how the aggregate bandwidth of swarms evolves as we increase the data center bandwidth budget and Fig. 11b presents the speed up gains of our strategy using the equal sharing strategy as a baseline. Our mechanism outperform equal and proportional sharing by a factor of x55 and x45 respectively in the best case when the data center bandwidth is scarce. These differences are reduced as the seeder bandwidth becomes less congested. We can observe as well that our solution outperforms the AntFarm strategy no matter the data center bandwidth, also achieving a more stable aggregate throughput in steady state. The AntFarm variability comes from the fact that the small increments in their initial phase not always awards bandwidth to the swarm which will obtain a higher benefit because of innacurate measurements.

The difference between these approaches comes from the way different swarms compete for the same bandwidth bottleneck. Our strategy is able to differentiate which swarms will benefit the most from the limited supply of data center capacity by granting more bandwidth to those swarms which will be able to redistribute more efficiently blocks of content to peering participants, whereas the other strategies make no distinction among swarms and award capacity as long as their downstream capacity is not saturated.

Besides the maximum aggregate bandwidth obtained by each mechanism, we also evaluated the convergence time. Fig. 11c shows the evolution of the aggregate bandwidth of swarms for different strategies during a simulated period of 10000 seconds when the data center bandwidth is 2048kbps. We can see that the *smart* seeding strategy as well as *equal* and *proportional* sharing converge to a steady state in few minutes. This is because all available



(a) Aggregate BW

(b) Speedup

(c) Aggregate BW Evolution

(d) Simulation vs PlanetLab

Figure 11: Experimental evaluation. Our smart seeding strategy outperforms other strategies by simulation.

datacenter bandwidth is split among different swarms –according to the different strategies– at the beginning of the measurement period. In contrast, the Antfarm strategy takes longer time to converge to a steady state –an order of magnitude 6 times higher– because of the slow initial phase in which it awards seeder bandwidth in small increments until all available bandwidth is allocated. This slow initial phase would worsen as the available data center bandwidth is even higher.

If we consider the benefits from the user experience point of view, the faster convergence time jointly with the higher throughput achieved by the *smart* seeding strategy is translated into lower download times for clients compared to the other strategies evaluated. This conclusion about the relation between the initial start up phase and download times is confirmed by the work of Sharma et al. [51].

Finally, to validate our simulations with deployed a real prototype using the planetary-scale testbed PlanetLab. The setup consisted in eight different swarms (sharing a single file each one) of membership sizes of 14, 5, 2 and 5 singleton swarms. We limited the upload bandwidth of peers to 50 kbps and the seeder bandwidth capacity was limited to 128 kbps. Using this setup, we compared our smart seeding strategy with the equal sharing strategy to asses the performance gains for around 30 minutes. As shown in Fig. 11d, the simulation shows that our smart seeding strategy outperforms by a factor of 3x in this specific scenario. We confirm our simulation results by comparing them to the results obtained from an experiment with real nodes on PlanetLab using the same setup (solid lines), which faithfully match our simulations (dashed lines).

References

[1] 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.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

[21] 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.

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

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

[24] 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.

[25] 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.

[26] 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.

[27] 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.

[28] 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.

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

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

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

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

[33] 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.

[34] 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

[35] 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

[36] 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.

[37] 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.

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

[39] S. Liu, X. Huang, H. Fu, and G. Yang, "Understanding data characteristics and access patterns in a cloud storage system," in Cluster, Cloud and Grid Computing (CCGrid), 2013 13th IEEE/ACM International Symposium on, May 2013, pp. 327–334.

[40] 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.–.

[41] 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.

[42] 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

[43] "Ubuntu one file services," http://one.ubuntu.com/.

[44] I. Drago, E. Bocchi, M. Mellia, H. Slatman, and A. Pras, "Benchmarking personal cloud storage," in Proceedings of the 2013 conference on Internet measurement conference. ACM, 2013, pp. 205–212.

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

[46] M. Feldman, K. Lai, and L. Zhang, "The proportional-share allocation market for computational resources," Parallel and Distributed Systems, IEEE Transactions on, vol. 20, no. 8, pp. 1075–1088, 2009.

[47] X. León and L. Navarro, "A stackelberg game to derive the limits of energy savings for the allocation of data center resources," Future Generation Computer Systems, vol. 29, no. 1, pp. 74 – 83, 2013. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0167739X12001306

[48] X. Leon, R. Chaabouni, M. Sanchez-Artigas, and P. Garcia-Lopez, "Smart cloud seeding for bittorrent in datacenters," Internet Computing, IEEE, vol. 18, no. 4, pp. 47–54, July 2014.

[49] B. Chun, D. Culler, T. Roscoe, A. Bavier, L. Peterson, M. Wawrzoniak, and M. Bowman, "Planetlab: an overlay testbed for broad-coverage services," ACM SIGCOMM Computer Communication Review, vol. 33, no. 3, pp. 3–12, 2003.

[50] R. Rahman, M. Meulpolder, D. Hales, J. Pouwelse, D. Epema, and H. Sips, "Improving efficiency and fairness in p2p systems with effort-based incentives," in Communications (ICC), 2010 IEEE International Conference on. IEEE, 2010, pp. 1–5.

[51] A. A. R. Abhigyan Sharma, Arun Venkataramani, "Pros & cons of model-based bandwidth control for client-assisted content delivery," in Proceedings of the 6th International Conference on Communication Systems and Networks, ser. COMSNETS'14, 2014.