CHARON: A Secure Cloud-of-Clouds System for Storing and Sharing Big Data

Ricardo Mendes, Tiago Oliveira, Vinicius Cogo, Nuno Neves, and Alysson Bessani

Abstract—We present CHARON, a cloud-backed storage system capable of storing and sharing big data in a secure, reliable, and efficient way using multiple cloud providers and storage repositories to comply with the legal requirements of sensitive personal data. CHARON implements three distinguishing features: (1) it does not require trust on any single entity, (2) it does not require any client-managed server, and (3) it efficiently deals with large files over a set of geo-dispersed storage services. Besides that, we developed a novel Byzantine-resilient data-centric leasing protocol to avoid write-write conflicts between clients accessing shared repositories. We evaluate CHARON using micro and application-based benchmarks simulating representative workflows from bioinformatics, a prominent big data domain. The results show that our unique design is not only feasible but also presents an end-to-end performance of up to $2.5\times$ better than other cloud-backed solutions.

Index Terms—Big-data storage, Cloud storage, Byzantine fault tolerance.

1 INTRODUCTION

The high volume, velocity, and variety of data being produced by diverse scientific and business domains challenge standard solutions of data management, requiring them to scale while ensuring security and dependability. A fundamental problem is where and how to store the vast amount of data that is being continuously generated. Private infrastructures are the first option for many organizations. However, creating and maintaining data centers is expensive, requires specialized workforce, and can create hurdles to sharing. Conversely, attributes like cost-effectiveness, ease of use, and (almost) infinite scalability make public cloud services natural candidates to address data storage problems.

Unfortunately, many organizations are still reticent to adopt public cloud services. First, few tools are already integrated with clouds, introducing difficulties to non-technical users. Second, as with most organizations dealing with critical information, there are concerns about trusting data to externally-controlled services that occasionally suffer from unavailability and security incidents [1], [2], [3], [4]. Finally, depending on the nature of the data being analyzed, there are legal restrictions impeding such institutions to outsource the storage and manipulation of some of the datasets, especially when involving personal information [5], [6].

We present CHARON, a near-POSIX cloud-backed storage system capable of storing and sharing big data with minimal management and no dedicated infrastructure. The main motivation for building this system was to support the management of genomic data, as required by bioinformatics and life sciences organizations [7].

As an example, consider the case of biobanks [8]. These institutions were originally designed to keep physical samples that could be later retrieved for research purposes. More recently, they are becoming responsible also for storing and analyzing the data related to such samples [9]. A sequenced human genome can reach up to 300GB, and each individual may have his genome sequenced many times during his life. The problem is that *biobanks lack the scalable infrastructure for storing and managing this potentially vast data volume*. Public clouds have plenty of resources for that.

Furthermore, the use of widely-accessible cloud services would facilitate the sharing of data among biobanks, hospitals, and laboratories, serving as a managed repository for public and access-controlled datasets. This would enable research initiatives that are not possible today due to the lack of a sufficient number of samples in a single institution [10], [11]. For example, 20–50k samples are required to study the interactions between genes, environment, and lifestyle that enable (or inhibit) a complex disease [12]. The rarer a disease is, the longer it takes to gather all necessary samples [10]. Given the rarity of some diseases, it is unlikely that a single hospital or research institute will ever be able to collect the required number of samples. The problem is *how to exploit the benefits of public clouds for data storage and sharing without endangering the security and dependability of biobanks' data.*

CHARON uses cloud-of-clouds replication [13], [14], [15], [16] of encrypted and encoded data to avoid having any cloud service provider as a single point of failure, operating correctly even if a fraction of the providers are unavailable (which happens more often than one would expect [1], [4]) or misbehave (due to bugs, intrusions, or even malicious insiders [2]). This ensures, for instance, the *security-by-design* requirement of Europe's General Data Protection Regulation (GDPR) [5]. Moreover, to further comply with data protection legislation, CHARON allows storing datasets in distinct locations (cloud-of-clouds, single cloud, or private repository) under the same namespace, exploring different tradeoffs on compliance, guarantees, performance, and costs.

Another distinguishing feature of CHARON is its *data-centric design*, in the sense that it does not depend on any custom server or code running on the cloud, relying instead only on cloud-managed services (e.g., Amazon S3 [17] or Azure Queue [18]).

- *All authors are with LASIGE, Faculdade de Ciências, Universidade de Lisboa, Lisboa, Portugal. Authors e-mails: rmendes@lasige.di.fc.ul.pt, toliveira@lasige.di.fc.ul.pt, vielmo@lasige.di.fc.ul.pt, nfneves@ciencias.ulisboa.pt, and anbessani@ciencias.ulisboa.pt.*

- *This work was supported by European Commission through projects BiobankCloud (FP7-ICT-317871), SUPERCLOUD (H2020-ICT-643964), and DiSIEM (H2020-DS-700692) and by FCT through projects IRCoC (PTDC/EEISCR/6970/2014), and LASIGE Research Unit (UID/CEC/00408/2019).*

This brings two important benefits. First, *cost savings* are expected since executing servers in cloud VMs is more expensive than resorting to cloud-data services. Second, a *significant reduction in management tasks* because our *data-centric design* removes the managing effort of keeping servers operating properly in cloud VMs [19].

Dropbox and similar file synchronization services do not solve the problems CHARON addresses since they require trust on a single provider/company. This limitation has been addressed by recent works on multi-cloud file synchronization services [20], [21], [22], but none of them targets big data or supports diverse data locations (see other differences in §6). Besides that, cloud-of-clouds storage systems either *have little or no support for controlled sharing* [13], [23], *rely on custom servers running on the cloud* [24], [25], or *provide only a programming library exporting read/write registers* (roughly equivalent to a disk block) [14], [15], [16], which are simpler and insufficient for implementing a fully-fledged big-data enabled distributed file system.

CHARON employs a set of Byzantine-resilient data-centric algorithms [15], [26], including a novel leasing protocol to avoid write-write conflicts on shared files. This protocol allows the implementation of strong synchronization guarantees without relying on custom servers (e.g., as in SCFS [24]) or *assuming linearizable object storage services*. For instance, DepSky's mutual exclusion [15] and MetaSync's pPaxos [21] rely on the strong consistency of such services. However, this property does not hold when listing the objects in a container in popular services like Amazon S3 [27] and Rackspace Cloud Files [28], leading to potential safety violations in these algorithms if such services are used.

Furthermore, CHARON is capable of handling big data in a secure way by dividing files into chunks, employing encryption, erasure codes, and compression, and using prefetching and background uploads. The way we integrate these techniques into a usable system makes CHARON unique, both in terms of design and offered features. Furthermore, the end-to-end performance of CHARON is 2–4× better than competing multi-cloud systems (e.g., [15], [20]), offering a usage experience as good as standard NFS.

In summary, the paper contributions are:

- The design and implementation of CHARON, a *practical* cloud-backed storage system for storing and sharing big data (§2 and §4);
- A Byzantine-resilient data-centric lease algorithm that exploits different cloud services without requiring trust on any of them individually (§3);
- An evaluation comparing CHARON with local, networked, and cloud-backed storage systems, using microbenchmarks and a novel benchmark that captures the I/O of bioinformatics applications (§5).

The Supplemental Material provides formal descriptions and correctness proofs for all lease algorithms described in the paper and an extensive description of the bioinformatics benchmark.

2 CHARON'S DESIGN

2.1 System and Threat Model

Like in previous works (e.g., [15], [21], [26]), we consider an unconstrained set of clients and a group of cloud providers. Each client has a unique id, an account for each cloud, and limited local storage. Every cloud provider offers one or more services, which implement access control mechanisms to ensure that only authorized accounts can access them.

To minimize the trust assumptions in the system, we consider that an unbounded number of clients and up to f cloud services can experience arbitrary (or Byzantine) faults. As in most storage systems, *malicious clients can jeopardize the security properties of the files they have access to: they can leak information about the data they can read and modify or destroy the data they can write*. Furthermore, the security properties of all information stored in faulty parties (clients or clouds) can be also compromised. As part of our assumptions, we include the extreme scenario where an active adversary takes control of all faulty parties, making them act together maliciously.

CHARON implements a security model where the *owner of the file pays for its storage* and defines its permissions. This is enforced by mapping the file system permissions (POSIX ACLs) to cloud services access control mechanisms (see details in §4.3). Therefore, a malicious client can only see, modify, and delete its own files and the files shared with him.

2.2 Design Overview

CHARON is a distributed file system that provides a near-POSIX interface to access an ecosystem of multiple cloud services and allows data transfer between clients. The preference for a POSIX interface rather than using data objects resorts to the fact the envisioned users are likely to be non-experts, and existent life sciences tools use files as their input most of the times. In particular, the system needs to (1) efficiently deal with multiple storage locations, (2) support reasonably big files, and (3) offer controlled file sharing. These challenges are amplified by our goals of excluding user-deployed servers and of requiring no modifications to existing cloud services (for immediate deployability).

Addressing these challenges requires a *novel data-centric protocol to coordinate metadata writes*, and the adaptation of several multi-cloud storage techniques for working in a practical setting. The demand for these novel designs comes from the lack of efficient solutions to fulfill our needs (i.e., data-centric design, dependability, and performance).

All techniques used in CHARON were combined considering two important design decisions. First, the system absorbs file writes in the client's local disk and, in the background, uploads them to their storage location. Similarly, prefetching and parallel downloads are widely employed for accelerating reads. This improves the usability of CHARON since transferring large files to/from the clouds take significant time (see §5). Second, the system avoids write-write conflicts, ruling out any optimistic mechanism that relies on users/applications for conflict resolution [20], [29], [30]. The expected size of the files and the envisioned users justify this decision. More specifically, (1) solving conflicts manually in big files can be hard and time-consuming; (2) users are likely to be non-experts, normally unaware of how to repair such conflicts; and (3) the cost of maintaining duplicate copies of big files can be significant. For instance, collaborative repositories, such as the Google Genomics [31], require such control since they allow users to read data about available samples, process them, and aggregate novel knowledge on them by sharing the resulting derived data into the bucket containing the sample of interest.

CHARON *separates file data and metadata* in different objects stored in diverse locations and manages them using different

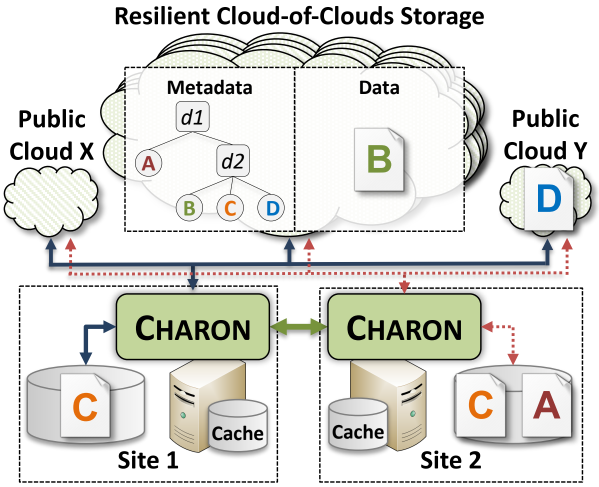


Figure 1. CHARON architecture.

strategies, as illustrated in Figure 1. File data locations are of three kinds in CHARON: cloud-of-clouds, single (public) storage cloud, and private repository (e.g., a private cloud). These alternatives explore various cost-dependability tradeoffs and address all placement requirements we have encountered with life sciences and big data applications. For example, the cloud-of-clouds can store critical data (CHARON's namespace and file B) that needs the availability and confidentiality assured by the multi-cloud scenario (provider-fault tolerance). A single cloud can store non-critical public studies and anonymized datasets (file D) (provider-dependent and potentially less expensive). Finally, private repositories must be used to keep clinical data from human samples that cannot leave the boundaries of a particular institution (file A) or country (file C) (subject to local infrastructure restrictions).

CHARON maintains the namespace tree, together with the files' metadata, replicated in the cloud-of-clouds storage. The rationale for this decision is to keep the file system structure secure by exploiting the expected high availability of cloud-of-clouds [15] and by expanding on the efficient data-centric replication protocols developed in the last years [14], [15], [16], [26]. The objective is to have *only soft state in clients*, which can be reconstructed after a crash by fetching data from the clouds.

In a very high level, CHARON interacts with the clouds for three main reasons: (1) storing/retrieving files' data, (2) storing/retrieving file system's metadata, and (3) obtaining/releasing leases to avoid write-write conflicts. The order of these interactions depends on the operation a client is performing. In a write, the client obtains the lease, uploads the files' data, and uploads the corresponding metadata. In a read, the client obtains the file system's metadata and then downloads the data associated with the requested files. The next section describes the computing abstractions these interactions rely on, while Section 4 details how the described operations works.

2.3 Building Blocks

CHARON is designed around three distributed computing abstractions built on top of basic cloud services. One of its key innovations is how it combines these abstractions to manage and share big data in a *principled and practical* way (see §4).

Every file chunk is written as a *non-updatable object* called **one-shot register**.[1] When a file is modified, a new chunk is created and pointed in the file metadata, independently of the file chunk location. The *integrity* of a file is attained by cross-verifying the validity of the stored chunks using SHA-256 cryptographic hashes saved in all clouds. *Confidentiality* is enforced by encrypting the data with a randomly generated AES 256-bit length key and storing it in a secure cloud-based single-writer multi-reader register (see the next building block). If the file is to be stored in a single cloud or private repository, the chunks are just written there. Otherwise, in cloud-of-clouds storage scenario, every encrypted chunk is encoded in $3f+1$ blocks through a storage-optimal erasure code [33]. The encoded blocks are then distributed among a quorum of $2f+1$ different cloud storage services (e.g., Amazon S3 [17]), in the best case scenario, to improve *availability*. In this case, to reconstruct the encrypted file chunk, a client must combine $f+1$ of such blocks. This reduced *read quorum* is sufficient because one-shot registers are non updatable, so there is no need to differentiate versions of the data. Overall, the storage overhead of this mechanism is only 50% more than when keeping blocks on a single cloud (for $f=1$), but different values can be obtained by playing with erasure code parameters and using more clouds. Both in read and write operations, this register requires only one call to each cloud.

Metadata objects representing shared folders can be read by multiple clients while being updated. This means that a one-shot register cannot implement them, requiring thus a **single-writer multi-reader (SWMR) register**. To address this, we have resorted to DepSky [15], which implements a data replication solution that offers a regular SWMR register abstraction on top of $3f+1$ cloud storage services. We redesigned and optimized the DepSky solution to support parallel cloud connections and to build a metadata service on top of it (see §4.1). The regular semantics means that readers operating concurrently with a writer can get both the metadata version being written or the previous one. This is achieved by using f-dissemination Byzantine quorums [34] for establishing the most recent version of a small metadata file with the version number and the cryptographic hash of the data signed by the writer. The approach to enforce the security properties of the stored data has some similarities with one-shot registers. The fundamental differences are: (1) the encryption key is split into shares (using Shamir's secret sharing [35]) that are stored with each encoded block [36], and (2) read/write operations require two cloud accesses (one for the metadata file, and another for the data itself). The first mentioned difference ensures that a client who has been granted access to a shared folder (see §4.3) will be able to regenerate the key to decrypt the metadata object. Consequently, he can obtain the pointers and encryption keys from the metadata object to access data stored in one-shot registers.

To preclude write-write conflicts on shared directories, CHARON relies on an efficient **leasing object** that tolerates Byzantine faults. This novel object, which is fundamental for ensuring a sub-second lock acquisition latency, is detailed in the next section.

In summary, CHARON uses one-shot registers to store the system's data, SWMR registers to store the system's metadata, and leasing objects to assign leases to shared directories for avoiding write-write conflicts.

[1] In the distributed computing literature, a *register* [32] is a shared memory object that offers two operations (read and write) satisfying the following sequential specification: every read returns the last written value.

3 Leases in the Cloud-of-Clouds

Leases are time-based contracts used to control concurrent accesses to resources while preventing version conflicts [37]. In this section, we describe the novel Byzantine-resilient leasing algorithms we use to avoid write-write conflicts in CHARON. Besides its resilience to Byzantine failures, a defining innovation of our construction when compared to well-known lease algorithms used in practice (e.g., the ones on top of Apache Zookeeper [38]) is our data-centric design: we require no custom code deployed on the clouds. The formalization and correctness proofs of these algorithms are present in the Supplemental Material.

3.1 Model and Guarantees

The lease algorithm follows the system and threat model stated in §2.1), with a few additional assumptions.

In this section, we refer to cloud services as *base objects*. A client may invoke, on the same base object, several parallel operations that are executed in FIFO order. Since a lease implies timing guarantees, an upper bound interval is assumed for the message transmission between clients and base objects. However, *this assumption is required only for liveness* since safety is always guaranteed.

Our algorithm ensures that at most one correct client at a time can access the shared resource and only for a limited duration (the *lease term* [37]). Each resource provides three lease-related operations: `lease(T)` and `renew(T)` acquires and extends the lease for T time units, respectively, while `release()` ends the lease. These operations satisfy the following properties:

- *Mutual Exclusion (safety):* There are never two correct clients with a valid lease for the same resource.
- *Obstruction-freedom (liveness):* A correct client that tries to lease a resource without contention will succeed.
- *Time-boundedness (liveness):* A correct client that acquires a lease will hold it for at most T time units unless the lease was renewed.

Our algorithm satisfies obstruction-freedom instead of stronger properties to pursue better performance in contention-free executions since write-contention is expected to be infrequent. Additionally, the time-boundedness property ensures that, if a client does not release a valid lease (e.g., due to a crash), the lease will be available again after at most T time units.

3.2 Byzantine-Resilient Composite Lease

Previous data-centric fault-tolerant mutual exclusion algorithms (e.g., [15], [39]) were designed to work directly on top of storage services. In this paper, we propose a more modular approach in which we build non-fault-tolerant *base lease objects*, each on top of a specific cloud-provided service, and $3f+1$ of these services are combined in an f-fault-tolerant *composite lease object*. This approach allows the design of more efficient base lease objects on top of any cloud-provided service (e.g., queues) instead of relying on fault-tolerant register constructions as in previous works [26], [39]. We still provide the design of base lease objects on top of storage services because they are the only abstraction available in certain cloud providers (e.g., [40]).

The lease operation of the composite lease object appears in Algorithm 1. To acquire a composite lease, a client simultaneously calls `lease` in each of the $3f+1$ base lease objects (Lines 5–6) and waits for either $2f+1$ successes or $f+1$ failures (Line 7).

ALGORITHM 1: Composite resource leasing by client *c*.

```
1  function $lease(leaseDuration)$ begin
2      $result \leftarrow nok$;
3      repeat
4          $L[0 .. 3f] \longleftarrow \perp$;
5          parallel for $0 \leq i \leq 3f$ do
6              $L[i] \longleftarrow baseLease_i.lease(leaseDuration)$;
7          wait until $|\{i : L[i] = ok\}| > 2f \vee |\{i : L[i] = nok\}| > f$;
8          if $|\{i : L[i] = ok\}| > 2f$ then
9              $result \leftarrow ok$;
10         else
11             for all $i : (L[i] = \perp) \vee (L[i] = ok)$ do
12                 $baseLease_i.release()$;
13             sleep for some time;
14     until $result \neq nok \vee$ $operation\ times\ out$;
15     return $result$;
```

In the first case, the client acquired the lease. Otherwise, the lease is unavailable or under contention, and the client needs to release all potentially obtained leases (the successful and the unanswered ones) and backoffs (Lines 11–13). This algorithm is repeated until it succeeds or a timeout is triggered.

Releasing a lease requires invoking the `release` operation in all base objects (as in Lines 11–12). The `renew` algorithm is similar to the one for `lease`, but with one additional cloud access to remove the old version of the lease being renewed. These two operations (`release` and `renew`) are never executed in the critical path of CHARON (see §4.1), and thus have little effect on the latency of the system.

3.3 Base Lease Implementations

Most public cloud services, from object storage to atomic database-as-a-service, provide basic features to create base lease objects. However, every implementation of a base lease object must comply with some specifications to work together in a composite lease. First, the `lease` operation requires the successful creation of (various flavors of) lease entries in the cloud service. Second, lease entries must be signed before they are sent to the cloud to ensure that cloud providers cannot create or corrupt leases. Third, base lease implementations rely on the access control from cloud services to guarantee that only authorized clients can access a lease. In doing so, malicious clients can only hinder correct users that inadvertently gave them access to a shared resource, but never affect resources shared among correct clients. Fourth, clients do not assign local timestamps to the lease to enforce the lease validity time. Instead, they use either the mechanisms provided by some services (e.g., augmented queues) that allow specifying the duration of lease entries or they mark the lease entries with the lease duration using the timestamps in the replies from cloud services. These timestamps are also used to check lease validity (instead of local clocks). This moves the need for bounded clock drifts from clients to cloud providers, which are expected to have much better time sources. In the following, we describe the implementation of four base lease objects using different cloud services.

Object storage services keep variable-length data objects in containers accessible through a hierarchical key-value store interface. Our lease object for these services works in three steps. A client starts by listing the objects in the container associated with the aimed resource. If no valid lease entries were found, it inserts a new signed entry in the container and lists the objects again to check if other entries were inserted concurrently. If another valid lease entry was observed in any of the two list operations, the client removes its entry and returns *nok*. Otherwise, the leasing succeeds.

This algorithm has been implemented in services like Google Storage [41], Azure Blob Storage [42], Rackspace Files [43], and Amazon S3 [17] since most of them guarantee strong consistency when creating objects and provide a timestamp on every service reply. However, the fact that listing objects from a container does not satisfy strong consistency in some services [27], [28] makes constructions based on them susceptible to safety violations.

Augmented queues such as Windows Azure Queue [18] and Rackspace Queue [44] have strongly-consistent enqueue, dequeue, and list functions, providing thus a universal shared memory abstraction capable of solving synchronization problems [45]. These queue services permit clients to specify the duration in which the lease entry is valid. After that time, the entry vanishes from the queues, releasing the lease.

In this algorithm, a client lists the queue to check if there are entries from other contenders. If the queue was empty, it enqueues a signed lease entry and lists the queue again to check if its entry is the valid one with the lowest index (queue head), which means the leasing succeeds. If the first list operation returns at least one valid entry from another client, the client returns *nok*. However, if the existing valid entry belongs to it (e.g., when renewing), the client pushes a new signed entry and removes all older ones to let the new lease be at the head of the queue.

NoSQL databases store data as pairs containing a unique key associated with a set of values. Amazon DynamoDB [46] provides a strongly-consistent service with a conditional update operation that enables the implementation of efficient base lease objects.

In this algorithm, a client verifies if an entry for the aimed resource already exists in the database. If there is an entry and it belongs to another client, then the operation returns *nok*. Otherwise, the client writes the new signed lease entry with the conditional update operation, which ensures the entry is only set if no other client added an equivalent entry in the meanwhile, and returns the result of this operation.

Transactional databases store data in tables and support ACID transactions. Google Datastore [47] is a cloud-based transactional database-as-a-service, which allows the implementation of base lease objects.

In this algorithm, the client queries the database for a lease entry about the target resource. If there is a valid entry belonging to another client, the transaction is aborted, and the operation returns *nok*. Otherwise, the client writes a new signed lease entry and commits the transaction. If the commit succeeds, the lease is obtained. Otherwise, conflicts were detected, and the operation returns *nok*.

3.3.1 Comparing base lease objects

All of these algorithms satisfy the obstruction-freedom property. However, most algorithms that are not based on object storage satisfy also deadlock-freedom [48], which guarantees that if several clients concurrently try to acquire a lease, one of them will succeed. Regarding the number of cloud accesses, they require only two to four accesses for acquiring a lease. The costs of running the base lease objects, for moderate-contention scenarios, will be significantly cheaper than running a fault-tolerant lock service in multiple clouds' VMs. More specifically, each lease acquisition can cost μ\$18.6, while having four medium-sized VMs in different providers may cost \$113/month, plus the management effort. Table 1 compares the base objects discussed in this section.

Table 1
Base Lease Objects Built on Top of Cloud Services.

Service	#A	Costs (μ\$)	Progress
AWS S3	3	15	Obstruction-Freedom
Google Storage	3	15	Obstruction-Freedom
Azure Blob Storage	3	15.6	Obstruction-Freedom
RackSpace Files	3	0	Obstruction-Freedom
Azure Queue	3	1.2	Deadlock-Freedom
RackSpace Queue	3	3	Deadlock-Freedom
AWS DynamoDB	2	subs.∗	Deadlock-Freedom
Google Datastore	4	2.4	Obstruction-Freedom

The table shows the number of cloud accesses (#A) necessary to acquire a lease in the absence of contention; the monetary costs (in microdollars) of such operation; the progress property satisfied by each base lease object algorithm, either obstruction-freedom or deadlock-freedom (which is stronger). ∗This service is charged per month.

4 CHARON IMPLEMENTATION

CHARON is a user-space file system implemented using FUSE-J, a Java wrapper for the FUSE library [49]. The system is fully implemented at the client side, using cloud services for storage and coordination, and is publicly available as open-source software at https://github.com/cloud-of-clouds/charon-fs.

4.1 Metadata Organization

Metadata is the set of attributes assigned to a file/directory (e.g., name, permissions). Independently of the location of the data chunks, CHARON stores all metadata in the cloud-of-clouds using *single-writer multi-reader registers* to improve their accessibility and availability guarantees (see §2.3). More specifically, we redesigned and optimized the SWMR register implementation of DepSky [15] to improve the performance and concurrency as described in the remaining of this section.

4.1.1 Managing namespaces

All metadata is stored within namespace objects, which encapsulate the hierarchical structure of files and directories in a subdirectory tree. CHARON uses two types of namespaces: *personal namespace* (PNS) and *shared namespace* (SNS). A PNS stores the metadata for all non-shared objects of a client, i.e., files and directories that can only be accessed by their owner. Each client has only one associated PNS. On the other hand, a client has access to as many SNSs as the shared folders it can access. Each shared folder is associated with exactly one SNS, which is referenced in the PNSs of the clients sharing it.

Although similar, personal and shared namespaces differ in the way the hashes of the most recent versions of the files' data chunks are stored. These hashes are primarily used to validate the cached file chunks. In PNSs, these hashes are serialized together with the rest of files' metadata before they are stored in the clouds. In the case of SNSs, the hashes are stored in a separate *Chunk Hashes* (CH) object, explained in the next section. Another difference between a PNS and a SNS is that the latter is associated with a lease to coordinate concurrent write accesses between different users which must be acquired before any update is executed on a file or directory in the namespace. Since each SNS is associated

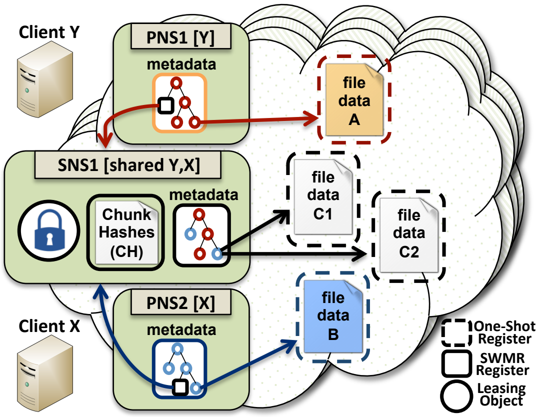


Figure 2. Objects maintained in the cloud by CHARON.

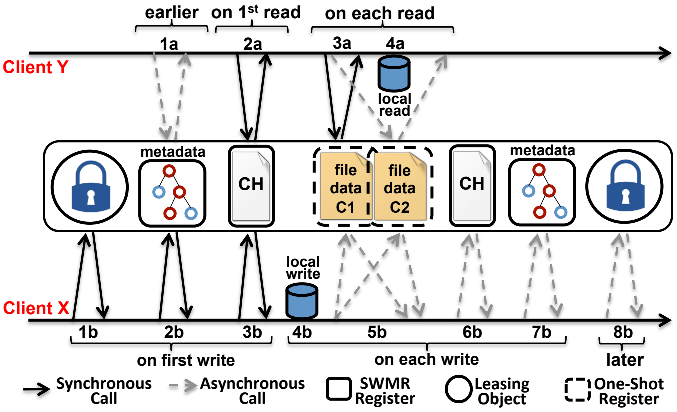


Figure 3. CHARON operation.

with one shared folder, a lease object is obtained to coordinate concurrent write accesses on an entire shared folder. Contrarily, concurrent readers do not require locking because the SWMR register used to store the metadata supports multiple readers even with concurrent writes.

Figure 2 depicts how a set of files relates to these namespaces. Files A and B are private to their owners (Clients Y and X, respectively). File C is divided into two chunks and is shared among these clients. Since files A and B are private, their metadata is kept in their owners' PNSs. In the case of file C, the reference to the file chunks is stored in SNS1.

4.1.2 Dealing with shared files

The PNS's metadata is downloaded from the cloud-of-clouds only once when the file system is mounted. SNSs, on the other hand, need to be periodically fetched to obtain metadata updates on shared directories. We opted to have a separate CH object in SNSs. Since a SNS contains all its files' metadata, having the hashes of every chunk of all files together with this information could significantly increase the monetary and performance costs of the periodic downloads. Therefore, the CH object is only refreshed when a file is opened (either for reading or write), to check if that file was updated. However, storing the hashes of all data chunks in a single CH object would be costly because the size of such object could grow linearly with the number of chunks referred by the SNS. To circumvent this problem, CHARON defines a maximum number of entries allowed in each CH object (e.g., 100 hashes). When this number is reached, newer chunks' hashes are saved in additional objects. There is an exception: hashes of chunks from the same file are always kept in a single CH to avoid fetching several CH objects when opening a large file.

Figure 3 illustrates what happens when Client X writes a file in a SNS while another client (Client Y) performs a concurrent non-cached read. When writing, Client X first must obtain a lease over the entire SNS (step 1b). The system then enforces the read of the metadata and CH objects (steps 2b–3b) to ensure that the most up-to-date metadata is used. In step 4b, the client writes the file in the local cache (see §4.2). After this point, all the steps are executed asynchronously. The changes it did are propagated to the adequate location (step 5b), the CH object and the SNS are updated in the cloud-of-clouds, and the lease is released (steps 6b–8b). This background propagation is crucial for the usability of the system since it can take a considerable time to complete. Notice that the steps 1b–3b are done only once during the first update on the SNS. After that, a client can update or create any file in that SNS while the lease is valid. CHARON uses a garbage collection protocol to delete the obsolete versions of data and metadata.

Client Y can concurrently perform read-only operations on the leased SNS while the client X is writing. As explained before, the SNS metadata is obtained from the clouds periodically. When performing the read operation, the client uses the current SNS metadata version (step 1a) and then reads the CH object containing the hashes of the most up-to-date version of each data chunk (step 2a). Next, if the file is not cached, the system downloads it to the cache (step 3a) and returns it to the client (step 4a). Since the metadata is uploaded only after the data is written by Client X, Client Y will never see data that belongs to unfinished writes.

4.2 Data Management

This section describes the most important techniques CHARON uses to manage big files efficiently.

4.2.1 Multi-level cache

CHARON uses the local disk to cache the most recent files used by clients. Moreover, it also keeps a fixed small main-memory cache to improve data accesses over open files. Both of these caches implement *least recently used* (LRU) policies. The use of caches not only improves performance but also decreases the operational cost of the system. This happens because cloud providers charge data downloads, but usually uploads are free as an incentive to send data to their facilities [50], [51], [52], [53]. It means that the CHARON operational cost is roughly the cost of the data storage plus the traffic necessary to download new versions of files.

4.2.2 Working with data chunks

Managing large files in cloud-backed file systems brings two main challenges. First, reading (resp. writing) whole (big) files from the cloud is impractical due to the high downloading (resp. uploading) latency [24]. Second, big files might not fit in the (memory) cache employed in cloud-backed file systems for ensuring usable performance [23], [24], [54], [55]. CHARON addresses these challenges by splitting (large) files into fixed-size chunks of 16MB, which results in blocks with a few megabytes after compression and erasure codes. This small size has been reported as having a good tradeoff between latency and throughput [15], [24], [55]. A chunk with few megabytes is relatively fast to load from disk to memory, can be transferred from/to clouds in a reasonable time, and is still

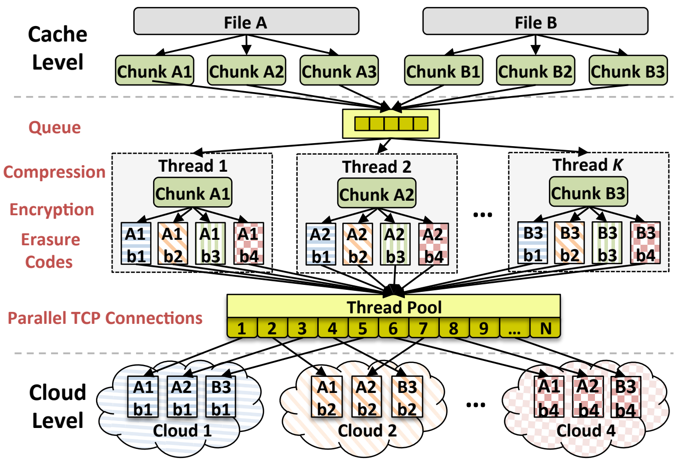


Figure 4. Data chunks management.

small enough to be maintained in main memory. Additionally, this approach is also cost-effective because, if a cached file is updated, only the modified data chunks need to be uploaded.

Figure 4 illustrates how CHARON manages and stores chunks in the clouds. After closing a file, all the new and updated chunks are inserted in a queue, from which they are consumed by a set of k threads. These threads are responsible for performing a sequence of compute-intensive operations on the chunks before uploading them to the clouds. First, we compress and encrypt the chunk. Next, we use a storage-optimal erasure code to generate $3f+1$ distinct coded blocks of the chunk, each with $\frac{1}{f+1}$ of its original size, in such a way that any set of $f+1$ blocks can be used to reconstruct it. After that, the threads use N workers on a thread pool to send the blocks to all the clouds. Blocks already have their cloud destination assigned when they arrive in these workers (as illustrated by the patterns in this figure). The threads consider a write operation complete only when they are notified that $2f+1$ blocks were successfully uploaded. We define a timeout for workers to finish these uploads to tolerate cases in which the clouds take too long to complete operations. If some worker does not perform some operation within the stipulated timeout, the thread using this worker cancels it and launches a new one to perform the same job.

In the case of a single cloud or a private repository is used, we do not employ erasure codes. Instead, after the chunk is encrypted and added to the thread pool, it is sent to the respective location by a single worker. In this case, if the storage location is unreachable, the data stays inaccessible.

Since we parallelize the transmission of all data blocks and wait for the first $2f+1$ clouds to complete, we always take advantage of the per-operation fastest clouds. This strategy aims to optimize the performance of uploading data to the clouds. It outperforms other systems' strategies in which they depend on the slowest clouds at some point [20], [22] or do not use techniques like compression or erasure coding [21]. Additionally, the configuration capability of all these architectural elements allows clients to adapt the behavior and bandwidth consumption of system's cloud-related operations with their own computing and network specificities. Moreover, employing preferred quorum techniques [56] in CHARON could adapt the system by first sending requests to clouds that best match user preferences beyond latency (e.g., cost, provider, geographical constraints).

4.2.3 *Prefetching*

The prominence of sequential reads in most big data workloads motivates the use of chunk prefetching. CHARON uses a thread pool for prefetching data chunks from any location as soon as a sequential read is identified. The system starts prefetching data when half of a chunk is sequentially read. If in the meanwhile the file being prefetched is closed, all its prefetching tasks are canceled. Besides reading data in advance, an additional advantage of this technique is to have several TCP connections receiving data in parallel, accelerating the download of the whole file.

4.3 Cloud-backed Access Control

CHARON implements a security model where the owner of the file pays for its storage and is able to define its permissions. This means that each client pays for all private data and all the shared data associated with the shared folders he created (independently on who wrote it). CHARON clients are not required to be trusted since access control is performed by the cloud providers, which enforce the permissions for each object. Moreover, the cloud-of-clouds access control is satisfied even if up to f cloud providers misbehave. This happens because if an object is read from up to f faulty providers, no useful information will be obtained (recall that data is encrypted and keys are stored using secret sharing in a SWMR register).

The implementation of this model requires a mapping between the file system and cloud storage abstractions. When configuring the system, users define the API credentials of the $3f+1$ clouds CHARON will use. In this way, when a CHARON client starts, it authenticates in each one of the providers, and after that each file or directory a user creates results in the creation of one or more objects associated with user cloud accounts. Sharing is allowed only at the granularity of directory subtrees. To give others the permission to access a directory, a user only needs to change the POSIX ACL associated with the desired directory. When this happens, CHARON transparently maps these permission changes in the clouds access control mechanisms using their APIs [57]. Namely, to share a directory, the system creates a SNS and gives permission to the grantee users, updating also the owner's PNS.

Each CHARON user identifier is mapped to the corresponding cloud accounts identifiers. This mapping is kept together with the client PNS in the cloud-of-clouds. Since there is no centralized server informing clients about the arrival of other clients to the system, the discovery of new clients and shared directories has to be done by external means such as mail invitation, as in Dropbox. For example, in a biobank federation, the biobanks must know the cloud account identifiers from each other.

5 EVALUATION

We evaluate CHARON and compare it with other systems. The experiments present results of (1) the latency of the leasing algorithms, (2) several microbenchmarks of metadata and data-intensive operations, and (3) a bioinformatics benchmark.

5.1 Experimental Environment

We used four machines (Intel Xeon E5520, 32 GB RAM, 15k-RPM HDD, SSD) connected through a gigabit network located in Portugal.

The three storage locations for CHARON were configured as follows. The *cloud-of-clouds storage* uses Amazon S3 (US), Windows Azure Storage (UK), Rackspace Cloud Files (UK), and Google Cloud Storage (US). For the *single cloud storage*, we use only Amazon S3 (US). The *private repository* was either located in the client's machine disk or in a different machine in the same LAN. For the *composite lease*, we use additional cloud services: Azure Queue [18], RackSpace Queue [44], Amazon DynamoDB [46], and Google Datastore [47]. Therefore, all cloud-of-clouds configurations consider $f = 1$.

We compare CHARON with the ext4 local file system, a Linux's NFSv4 deployed in our cluster, and other cloud-backed file systems with the code available on the web (e.g., SCFS [24] and S3QL [54]). CHARON and the other cloud-backed systems were configured to upload data to the clouds in the background. In the case of SCFS [24], the coordination service replicas were deployed in four medium VMs on Amazon EC2 (UK).

5.2 Composite Leasing

In this section, we evaluate the composite lease algorithm described in §3.2 and the base objects used in its implementation. We focus our analysis on the lease operation since it is the only one in the critical path of any application updating shared folders.

5.2.1 Contention-free executions

The composite leasing algorithm was configured in two ways: using only storage cloud services from different providers (ST), and using only non-storage cloud services, such as queues, from different providers (NST). Additionally, we compare these compositions with DepSky's mutual exclusion algorithm (DL) [15], using the same services as the ST configuration. Figure 5(a) presents the lease latency of these algorithms and their base lease objects.

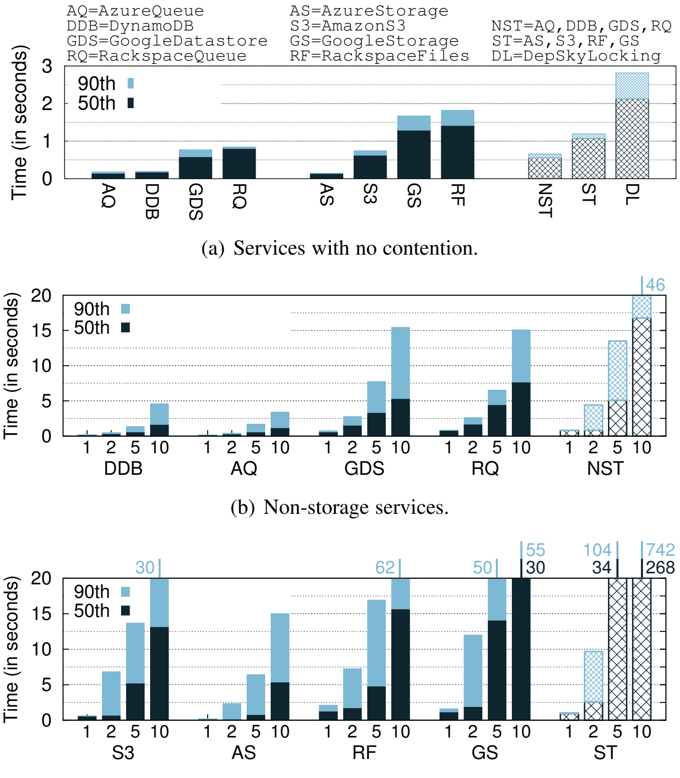
The base lease objects require between 0.2 to 1.8s to acquire a lease. Overall, the results for non-storage base lease objects are better than the ones using storage services. This happens probably because storage services are throughput-oriented, and thus less effective when dealing with small objects such as lease entries.

The results of the composite lease configurations reflect the performance of their base objects. More specifically, the composite lease protocol waits for a quorum of $2f+1 = 3$ leasing acknowledgments from different services, which means that the resulting latency is similar to the third fastest cloud service. For instance, the latencies of NST and GDS are similar (≈600 ms), which is worse than DDB and AQ, but better than RQ. For a pure storage-based lease (ST), we observed a lease latency 2× higher than NST. Consequently, we use the NST configuration in all CHARON experiments.

The observed latency for *DepSky locking (DL) is twice the latency of ST and four times higher than NST* (used in CHARON). This difference happens because the DepSky algorithm accesses the storage clouds in phases, and not by executing base lease algorithms in parallel. DepSky's and our object storage lease algorithms have an additional disadvantage when compared with other algorithms: their required strong consistency for storage is not always guaranteed [27], [28].

5.2.2 Executions under contention

An important aspect of a lease algorithm is how its performance degrades with contention. We perform experiments with a varying number of clients (1, 2, 5, and 10) trying to acquire a lease on the same SNS (and releasing it right after), and measure the time for a client to acquire the lease. For obstruction-free lease algorithms, we use a random backoff time of up to one second.



(a) Services with no contention.

(b) Non-storage services.

(c) Storage services.

Figure 5. Latency of lease acquisition algorithms without contention and under contention of up to 10 clients.

The results for several base lease objects and composite lease configurations appear in Figures 5(b) and 5(b). Again, non-storage services (Figure 5(b)) provide better/faster results than storage services (Figure 5(c)). This happens because most non-storage services satisfy the deadlock-freedom property, i.e., if several processes try to acquire the lease concurrently, some process will succeed [48]. This makes them more efficient when dealing with contention than the storage-based algorithms, which only implement obstruction-freedom.

The composite lease (NST and ST in the figures) also provides only obstruction-freedom and thus has a superlinear increase in the waiting time when a lease is obtained under contention. This is expected as CHARON is not optimized for scenarios with a large number of clients trying to update the same folder or file. The composite lease algorithm is fast with one or two contending clients but noticeably slower with 5 or 10. The same behavior is observed for DepSky locking (not shown), but with higher latency. For instance, *the 90th latency for two contending clients in DepSky is* 2× *and* 10× *higher than ST and NST, respectively.*

5.3 File System Microbenchmarks

This section compares CHARON with other file systems using the Filebench microbenchmark suite [58]. The first two experiments are focused on evaluating the performance of isolated system calls, while the third one measures the latency associated with interacting with the clouds for downloading and uploading data. The presented results do not consider the lease acquisition time.

Table 2
Metadata-intensive microbenchmark results (ops/s).

Operation	ext4	NFS	S3QL	SCFS	CHARON
Create	2618	192	105	2	485
Delete	1895	2518	486	4	1258
Stat	15299	20881	5995	9	12925
MakeDir	14998	16664	4242	14	13665
DeleteDir	11998	6785	950	5	8665
ListDir	18759	17426	604	6	9894

Table 3
Data-intensive microbenchmark results (MB/s).

Operation	ext4	NFS	S3QL	SCFS	CHARON
seqRead	215	211	214	200	194
randRead	210	205	207	191	186
seqWrite	125	125	10	17	36

5.3.1 Metadata-intensive operations

Our first experiment focuses on how well the system deals with metadata intensive operations when compared with other systems. Table 2 presents the number of operations per second for ext4 (on SSD), NFS, S3QL [54], SCFS [24] (discussed in §6), and CHARON for different operations on 0-byte files.

The results show that CHARON *has a performance mostly within the same order of magnitude of ext4 and NFS*, being slower mainly due to the overhead of FUSE and its Java wrapper. *When compared with other cloud-backed file systems,* CHARON *is significantly faster* because metadata updates are executed only in the disk, and later sent to the cloud. S3QL uses an SQLite local database for that, and SCFS accesses the cloud on every metadata operation.

5.3.2 Data-intensive operations

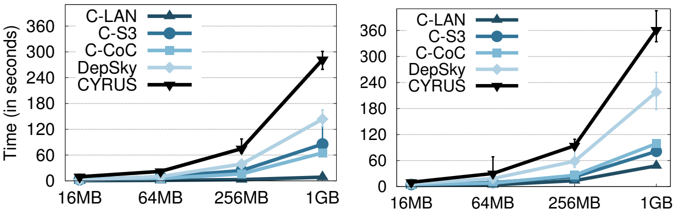
Table 3 presents the results for similar microbenchmarks, but now focusing on data-intensive operations with files of 256MB.

Unsurprisingly, ext4 offers the best read throughput both for sequential and random workloads. S3QL and NFS provide a slightly lower read throughput than ext4. Despite presenting a lower performance, SCFS and CHARON are still competitive for read workloads.

When considering write throughput, ext4 and NFS present the best performance for sequential workloads. However, CHARON *presents at least* 2× *better sequential write throughput than the other cloud-backed file systems*. In particular, *S3QL provides a write-throughput almost* 4× *lower than our system*. This happens because S3QL does not perform well when writing small chunks [59] (and the benchmark was configured with 8kB-writes).

5.3.3 Read and write of big files

Efficiency in reading and writing large files to/from the clouds is one of the main objectives of CHARON. Figure 6 shows the time required for sequentially read and write non-cached files (from 16MB to 1GB). We perform these experiments considering different data locations for CHARON, namely: private repository in the same network (C-LAN), single cloud in Amazon S3 (C-S3), and the cloud-of-clouds (C-CoC). Additionally, we also present values for two other multi-cloud data replication algorithms, DepSky [15] and CYRUS [20] (discussed in §6), using the same cloud services as C-CoC. We did not compare with other systems [21], [22] because they do not work in our environment. However, the results achieved for C-CoC are 80–325% better than the ones published for these systems. Differently from previous experiments, here the write latency includes the time necessary to upload the data to its final location.



(a) Non-cached read.

(b) Write and upload.

Figure 6. Non-cached read (download) and write (upload) latencies for different file sizes with CHARON (data in different locations), DepSky [15] and CYRUS [20].

When considering a cloud-of-clouds deployment, the results show that CHARON *(C-CoC) reads (resp. writes) 1GB files* 2× *and* 4× *(resp.* 2.2× *and* 3.5×*) faster than DepSky and CYRUS*, respectively. This happens because previous multi-cloud data replication algorithms are not optimized to deal with big files: they do not break the file in chunks (or use chunks too small [20]), neither use our upload strategy (see Figure 4), nor use techniques such as prefetching. Furthermore, CYRUS round-robin distribution of data chunks among all clouds (trying to balance storage usage) makes it quite slow, as some clouds are noticeably slower than others. On the other hand, our approach on data chunks management allows CHARON to perform as fast as the $f+1^{th}$ (resp. $2f+1^{th}$) fastest clouds when reading (resp. writing) chunks.

When comparing different data locations in CHARON, reading/writing from/to a private repository present the best latency, since the target location is inside our local network. The difference between the latency of CHARON using Amazon S3 or the cloud-of-clouds is quite small for both reading and writing results. For writing, the additional latency presented by the cloud-of-clouds comes from the fact that we need to write the data in three clouds to finish it. Thus, the end-to-end latency will be dictated by the third fastest cloud.

The main takeaway here is that *the use of Byzantine-resilient cloud-of-clouds storage does imply increased latency when compared with the use of a single cloud* and that CHARON *is significantly faster in dealing with big files than competing solutions* [15], [20].

5.4 Bioinformatics Workflows

Our last set of experiments aims to compare the performance of different configurations of CHARON and alternative systems using FS-Biobench, a novel storage benchmark in the domain of bioinformatics (described in the Supplemental Material). Other existing bioinformatics macro-benchmarks focus on CPU-bound tasks, use discontinued versions of broadly accepted tools, and overlook the I/O operations executed by these tools [60], [61], [62]. The FS-Biobench emulates specifically the I/O operations of eight representative bioinformatics workloads, summarized in Table 4, and is independent of any external tool. Most workflows use sequential reads and writes, and they differ in the number of accessed files, their size and structure, and the execution pattern

Table 4
Characteristics of the eight FS-Biobench workflows.

Workflow	Input Files	Output Files	Description
W1.Genotyping	–	0+1 (24MB)	Write a single genotyping file
W2.Sequencing	–	0+1 (1GB)	Write a single sequencing file in FASTQ format
W3.Prospection	2 (1MB)	0+1 (4kB)	Prospect appropriate samples for a study from two MIABIS XML files
W4.Alignment	1 (1GB)	0+1 (960MB)	Search DNA reads from `W2` in a reference, and write the alignment results
W5.Assembly	1 (1GB)	0+1 (18MB)	Write a contiguous DNA sequence from a FASTQ sequencing file
W6.GWAS	2 (48MB)	2+1 (432MB+200kB)	Read two genotyping files, perform a GWA study, and plot a graph
W7.Annotation	1 (1GB)	2+1 (1.07GB+268MB)	Align DNA reads, obtain genomic variations, and write an annotated VCF file
W8.Methylation	1 (1GB)	2+1 (999MB+4kB)	Align DNA reads, and write a list of methylated positions

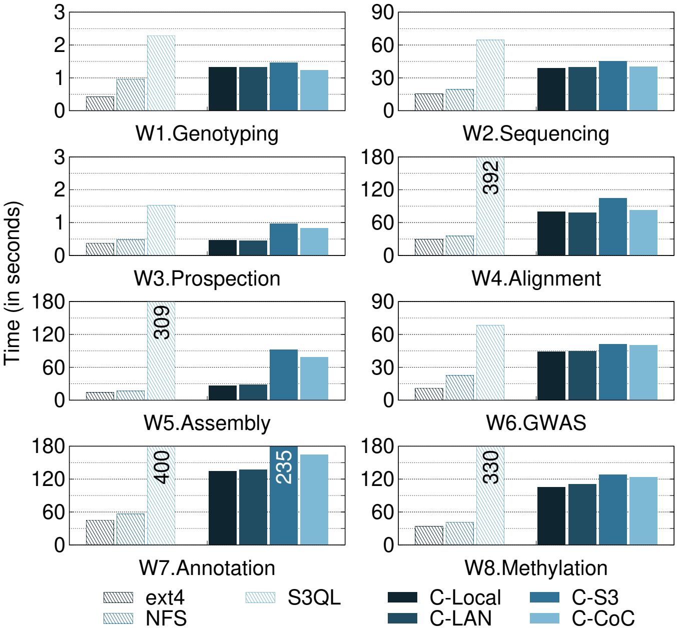


Figure 7. FS-Biobench execution for different configurations.

(e.g., reading an entire file before writing anything, interposing reads and writes, reading more than one file in parallel). These workflows include complex pipelines to achieve concrete results in bioinformatics and were selected from the workflows analyzed and implemented in the BiobankCloud project [7].

Figure 7 presents the duration of FS-Biobench workflows for ext4 on SSD, NFS with the client and server in the same LAN, S3QL, and CHARON using a repository in different locations: SSD in the same machine (C-Local), disk in a server in the same LAN (C-LAN), AWS S3 (C-S3), and cloud-of-clouds (C-CoC). SCFS is not evaluated because it does not support big files. We execute every workflow ten times on each scenario and report average values. CHARON's and S3QL's caches are cleaned after each workflow execution, as all results would be similar to C-Local if the files were cached.

Ext4 and NFS serve as a basis for comparison in this experiment and, as expected, are usually faster in running the workflows. The time needed for CHARON to finish workflows `W1` and `W2` are similar for all data locations. This happens because write operations immediately return after the file is updated in the local disk. Workflow `W3` lasts almost 2× longer to finish in C-S3 and C-CoC than in C-Local and C-LAN due to the latency of fetching the two small files from the remote cloud services. Workflows `W4`, `W5`, `W7`, and `W8` are the ones requiring more time to run since they need to read a 1GB FASTQ file from the repository. Workflow `W6` reads two genotyping files with only 24MB each, and thus requires less time to run. Nonetheless, workflows `W4`-`W8` rank the different data locations in the same order, where the C-Local is the fastest, followed by C-LAN, C-CoC, and C-S3. Interestingly, running the benchmark in a cloud-of-clouds-hosted repository has better results (compared to C-S3) due to the capability of fetching chunks from the two fastest clouds at the moment. More importantly, even considering that the latency of fetching input files dominate most of these benchmarks, in a real setup, the file processing can start as soon as the first chunk is available.

In conclusion, CHARON *(C-CoC) runs the workflows up to* 2.5× *(*`W4`*) faster than the other (single) cloud-backed file system (S3QL).* Furthermore, *our system using the cloud-of-clouds is* 30% *to* 200% *slower than NFS* in all workflows but `W5` (which is read-intensive). This is an excellent result as the latency of accessing the cloud is 100× higher than accessing a LAN-based server.

6 RELATED WORK

6.1 Distributed File Systems

CHARON is an intrusion-tolerant file system [63] that maintains data confidentiality, integrity, and availability despite the existence of compromised components. Our design adopts some ideas from existing file systems, such as the separation of data and metadata from NASD [64], volume leases from AFS [65], and background updates from several peer-to-peer file systems [66], [67], [68]. In particular, Farsite has some similarities with our system, but is crucially different in its use of Byzantine Fault-Tolerant (BFT) replica groups for assigning leases and maintaining metadata consistently [66]. Another related system is xFS [69], a data-centric network file system in which all data and metadata are stored at the client side.

A fundamental difference between these systems and CHARON is that in our solution clients interact using widely-available untrusted cloud services instead of communicating directly for coordination.

6.2 Data-centric coordination

A key feature of CHARON is the use of Byzantine-resilient data-centric algorithms for implementing storage and coordination. There are some works that propose the use of this kind of algorithms for implementing dependable systems [15], [26], [39].

Byzantine disk Paxos [26] is a consensus protocol built on top of untrusted shared disks. More recently, an enhanced version of this protocol specifically designed to use file synchronization services (e.g., DropBox, Google Drive) instead of disks was published [21]. These algorithms could be used to implement mutual exclusion satisfying deadlock-freedom (a stronger liveness guarantee than obstruction-freedom). However, these solutions would require a much larger number of cloud accesses. Our lease protocol, on the other hand, requires only two to four cloud accesses for acquiring a lease.

To the best of our knowledge, there are only two fault-tolerant data-centric lease algorithms in the literature [15], [39]. The lease algorithm of Chockler and Malkhi [39] has two important differences when compared with CHARON's BFT composite lease. First, it does not provide an always-safe lease as it admits the existence of more than one process with valid leases. Second, it tolerates only crashes, requiring thus some trust on individual cloud providers. The BFT mutual exclusion algorithm from DepSky [15] is a natural candidate to regulate access contention in CHARON. However, our composite lease algorithm is $4-10\times$ faster than DepSky's (see §5.2), does not require clients to have synchronized clocks, and neither rely on weakly-consistent operations such as object storage' list.

6.3 Multi-cloud storage

In the last years, many works have been proposing the use of multiple cloud providers to improve the integrity and availability of stored data [13], [14], [15], [16], [20], [22], [23], [24], [70]. A problem in some of them is the fact they only provide object storage (i.e., read/write registers), which hardens their integration with existing applications. Examples of these systems are RACS [13] for write-once/archival storage, and DepSky [15], its evolution [16], and ICStore [14] for updatable registers.

Systems like Hybris [23], SCFS [24] and RockFS [70] employ a hybrid approach in which unmodified cloud storage services are used together with few computing nodes to store metadata and coordinate data access. The main limitation of these systems is that they require servers deployed in the cloud providers, which implies additional costs and management complexity. The same limitation applies to modern (single-provider) geo-replicated storage systems such as Spanner [71], SPANStore [25] and Pileus [72], if deployed in multiple clouds.

A slightly different kind of work proposes the aggregation of multiple file synchronization services (e.g., DropBox, Box, Google Drive) in a single dependable service [20], [21], [22]. CYRUS [20] does not implement any kind of concurrency control, allowing different clients to create different versions of files accessed concurrently. UniDrive [22] employs a lock protocol based on quorums, in which a client has access to a shared folder as long as it is able to write a lock file alone in a majority of the services. However, it only tolerates crash failures and is not improved to handle big data. Finally, MetaSync's [21] main constraints are its assumption of a linearizable object storage service, which is not always the case [27], [28], and the use of full replication, which makes the download and upload of data slow and increases the monetary costs of the system. Regarding the latter, the way MetaSync order file updates does not make it easy to integrate erasure codes in the system. On the other hand, CHARON is the first data-centric cloud-backed file system to support the main requirements of life sciences and other big data domains.

7 CONCLUSIONS

CHARON is a cloud-backed file system for storing and sharing big data. Its design relies on two important principles: files metadata and data are stored in multiple clouds, without requiring trust on any of them individually, and the system is completely data-centric. This design has led us to develop a novel Byzantine-resilient leasing protocol to avoid write-write conflicts without any custom server. Our results show that this design is feasible and can be employed in real-world institutions that need to store and share large critical datasets in a controlled way.

REFERENCES

[1] Cloud Harmony, "Service Status," https://cloudharmony.com/status-of-storage-group-by-regions, 2019.

[2] Cloud Security Alliance, "Top Threats," https://cloudsecurityalliance.org/group/top-threats/, 2016.

[3] M. A. C. Dekker, "Critical Cloud Computing: A CIIP perspective on cloud computing services (v1.0)," European Network and Information Security Agency (ENISA), Tech. Rep., 2012.

[4] H. S. Gunawi *et al.*, "Why does the cloud stop computing?: Lessons from hundreds of service outages," in *Proc. of the SoCC*, 2016.

[5] European Commission, "Data protection," https://ec.europa.eu/info/law/law-topic/data-protection_en, 2018.

[6] G. Gaskell and M. W. Bauer, *Genomics and Society: Legal, Ethical and Social Dimensions*. Routledge, 2013.

[7] A. Bessani *et al.*, "BiobankCloud: a platform for the secure storage, sharing, and processing of large biomedical data sets," in *DMAH*, 2015.

[8] H. Gottweis *et al.*, "Biobanks for Europe: A challenge for governance," European Commission, Directorate-General for Research and Innovation, Tech. Rep., 2012.

[9] P. E. Verissimo and A. Bessani, "E-biobanking: What have you done to my cell samples?" *IEEE Security Privacy*, vol. 11, no. 6, pp. 62–65, 2013.

[10] P. R. Burton *et al.*, "Size matters: just how big is big? Quantifying realistic sample size requirements for human genome epidemiology," *Int J Epidemiol*, vol. 38, no. 1, pp. 263–273, 2009.

[11] D. Haussler *et al.*, "A million cancer genome warehouse," University of Berkley, Dept. of Electrical Engineering and Computer Science, Tech. Rep., 2012.

[12] R. W. G. Watson, E. W. Kay, and D. Smith, "Integrating biobanks: addressing the practical and ethical issues to deliver a valuable tool for cancer research," *Nature Reviews Cancer*, vol. 10, no. 9, pp. 646–651, 2010.

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

[14] C. Basescu *et al.*, "Robust data sharing with key-value stores," in *Proc. of the DSN*, 2012.

[15] A. Bessani, M. Correia, B. Quaresma, F. Andre, and P. Sousa, "DepSky: Dependable and secure storage in cloud-of-clouds," *ACM Trans. Storage*, vol. 9, no. 4, pp. 12:1–12:33, 2013.

[16] T. Oliveira, R. Mendes, and A. Bessani, "Exploring key-value stores in multi-writer Byzantine-resilient register emulations," in *Proc. of the OPODIS*, 2016.

[17] Amazon, "Amazon S3," http://aws.amazon.com/s3/, 2019.

[18] Microsoft, "Microsoft Azure Queue," http://azure.microsoft.com/en-us/documentation/articles/storage-dotnet-how-to-use-queues/, 2019.

[19] B. Martens, M. Walterbusch, and F. Teuteberg, "Costing of cloud computing services: A total cost of ownership approach," in *Proc. of the HICSS*, 2012.

[20] J. Y. Chung, C. Joe-Wong, S. Ha, J. W.-K. Hong, and M. Chiang, "CYRUS: Towards client-defined cloud storage," in *Proc. of the EuroSys*, 2015.

[21] S. Han *et al.*, "MetaSync: File synchronization across multiple untrusted storage services," in *Proc. of the USENIX ATC*, 2015.

[22] H. Tang, F. Liu, G. Shen, Y. Jin, and C. Guo, "UniDrive: Synergize multiple consumer cloud storage services," in *Proc. of the Middleware*, 2015.

[23] D. Dobre, P. Viotti, and M. Vukolic, "Hybris: Robust hybrid cloud storage." in *Proc. of the SoCC*, 2014.

[24] A. Bessani *et al.*, "SCFS: a shared cloud-backed file system," in *Proc. of the USENIX ATC*, 2014.

[25] Z. Wu, M. Butkiewicz, D. Perkins, E. Katz-Bassett, and H. V. Madhyastha, "SPANStore: Cost-effective geo-replicated storage spanning multiple cloud services," in *Proc. of the SOSP*, 2013.

[26] 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.

[27] Amazon, "Amazon S3 data consistency model," https://docs.aws.amazon.com/AmazonS3/latest/dev/Introduction.html#ConsistencyModel, 2019.

[28] Rackspace, "Cloud files - faqs," https://support.rackspace.com/how-to/cloud-files-faq/, 2019.

[29] J. J. Kistler and M. Satyanarayanan, "Disconnected operation in the Coda file system," *ACM Trans. Comput. Syst.*, vol. 10, no. 1, pp. 3–25, 1992.

[30] A. J. Mashtizadeh, A. Bittau, Y. F. Huang, and D. Mazières, "Replication, history, and grafting in the Ori file system," in *Proc. of the SOSP*, 2013.

[31] Google, "Google Genomics," https://cloud.google.com/genomics/, 2019.

[32] L. Lamport, "On interprocess communication (part II)," *Distributed Computing*, vol. 1, no. 1, pp. 203–213, 1986.

[33] J. S. Plank, "Erasure codes for storage systems: A brief primer," *;login: The USENIX magazine*, vol. 38, no. 6, pp. 44–50, 2013.

[34] D. Malkhi and M. Reiter, "Byzantine quorum systems," *Distributed Computing*, vol. 11, no. 4, pp. 203–213, 1998.

[35] A. Shamir, "How to share a secret," *Communications of ACM*, vol. 22, no. 11, pp. 612–613, 1979.

[36] H. Krawczyk, "Secret sharing made short," in *Proc. of the CRYPTO*, 1993.

[37] C. Gray and D. Cheriton, "Leases: An efficient fault-tolerant mechanism for distributed file cache consistency," in *Proc. of the SOSP*, 1989.

[38] P. Hunt, M. Konar, F. Junqueira, and B. Reed, "Zookeeper: Wait-free coordination for internet-scale services," in *Proc. of the USENIX ATC*, 2010.

[39] G. Chockler and D. Malkhi, "Light-weight leases for storage-centric coordination," *Int. J. Parallel Program.*, vol. 34, no. 2, pp. 143–170, 2006.

[40] Wasabi, "Wasabi Hot Cloud Storage," https://wasabi.com, 2019.

[41] Google, "Google storage," https://developers.google.com/storage/, 2019.

[42] B. Calder *et al.*, "Windows Azure storage: a highly available cloud storage service with strong consistency," in *Proc. of the SOSP*, 2011.

[43] Rackspace, "Rackspace cloud files," https://www.rackspace.com/cloud/files, 2019.

[44] ——, "Message queuing service with simple API - Rackspace cloud queues," http://www.rackspace.com/cloud/queues/, 2019.

[45] M. Herlihy, "Wait-free synchronization," *ACM Trans.Prog.Lang.Syst.*, vol. 13, no. 1, pp. 124–149, 1991.

[46] Amazon, "Amazon DynamoDB—NoSQL cloud database service," http://aws.amazon.com/dynamodb/, 2019.

[47] Google, "Google cloud datastore—NoSQL database for cloud data storage," https://cloud.google.com/datastore/, 2019.

[48] H. Attiya and J. Welch, *Distributed computing: fundamentals, simulations, and advanced topics*. John Wiley & Sons, 2004.

[49] FUSE, "File system in user space," https://github.com/libfuse/libfuse, 2019.

[50] Amazon, "Amazon S3 pricing," http://aws.amazon.com/s3/pricing/, 2019.

[51] Google, "Google storage pricing," https://developers.google.com/storage/docs/pricingandterms, 2019.

[52] Microsoft, "Windows Azure pricing," https://azure.microsoft.com/en-us/pricing/details/storage/blobs/, 2019.

[53] Rackspace, "Rackspace cloud files pricing," https://www.rackspace.com/cloud/files/pricing/, 2019.

[54] S3QL, "S3QL - a full-featured file system for online data storage," https://bitbucket.org/nikratio/s3ql/overview, 2019.

[55] M. Vrable, S. Savage, and G. M. Voelker, "BlueSky: A cloud-backed file system for the enterprise," in *Proc. of the FAST*, 2012.

[56] M. Abd-El-Malek, G. R. Ganger, G. R. Goodson, M. K. Reiter, and J. J. Wylie, "Fault-Scalable Byzantine Fault-Tolerant Services," in *Proc. of the SOSP*, 2005.

[57] T. Oliveira, R. Mendes, and A. Bessani, "Sharing files using cloud storage services," in *Proc. of the DIHC, co-located with Euro-Par*, 2014.

[58] Filebench, "Filebench webpage," https://github.com/filebench/filebench/wiki, 2019.

[59] S3QL, "S3QL 1.13.2 documentation: Known issues," http://www.rath.org/s3ql-docs/issues.html, 2016.

[60] K. Albayraktaroglu *et al.*, "Biobench: A benchmark suite of bioinformatics applications," in *Proc. of the ISPASS*, 2005.

[61] D. A. Bader, Y. Li, T. Li, and V. Sachdeva, "Bioperf: A benchmark suite to evaluate high-performance computer architecture on bioinformatics applications," in *Proc. of the IISWC*, 2005.

[62] Y. Li, T. Li, T. Kahveci, and J. Fortes, "Workload characterization of bioinformatics applications," in *Proc. of the MASCOTS*, 2005.

[63] J. S. Fraga and D. Powell, "A fault- and intrusion-tolerant file system," in *Proc. of the 3rd IFIP Conference on Computer Security*, 1985.

[64] G. Gibson *et al.*, "A cost-effective, high-bandwidth storage architecture," in *Proc. of the ASPLOS*, 1998.

[65] J. Howard *et al.*, "Scale and performance in a distributed file system," *ACM Trans. Comput. Syst.*, vol. 6, no. 1, pp. 51–81, 1988.

[66] A. Adya *et al.*, "Farsite: Federated, available, and reliable storage for an incompletely trusted environment," in *Proc. of the OSDI*, 2002.

[67] J. Kubiatowicz *et al.*, "OceanStore: An architecture for global-scale persistent storage," in *Proc. of the ASPLOS*, 2000.

[68] J. Stribling *et al.*, "Flexible, wide-area storage for distributed system with WheelFS," in *Proc. of the NSDI*, 2009.

[69] T. E. Anderson *et al.*, "Serverless network file systems," *ACM Trans. Comput. Syst.*, vol. 14, no. 1, pp. 41–79, 1996.

[70] D. R. Matos, M. L. Pardal, G. Carle, and M. Correia, "Rockfs: Cloud-backed file system resilience to client-side attacks," in *Proc. of the Middleware*, 2018.

[71] J. Corbet et. al, "Spanner: Google's globally distributed database," *ACM Trans. Comput. Syst.*, vol. 31, no. 3, pp. 8:1–8:22, 2013.

[72] D. B. Terry *et al.*, "Consistency-based service level agreements for cloud storage," in *Proc. of the SOSP*, 2013.

Ricardo Mendes received the MSc degree from the University of Lisbon, in 2012. He is currently a researcher at LaSIGE research unit and the Navigators research team. His main research interests include dependable and secure cloud storage systems. Over the years, he participated in several European and national research projects with a focus on dependability and security, such as TClouds, BiobankCloud and SUPERCLOUD.

Tiago Oliveira has a Master in Systems Architecture and Computers Network, from Faculdade de Ciências (ULisboa), specializing in data storage, cloud computing and systems' dependability. During the past years he has been involved in studying, designing, and developing cloud-based storage systems aiming to improve the reliability of the stored data.

Vinicius Cogo has a MSc in Informatics and is a PhD student from the Faculty of Sciences, University of Lisbon. He is member of the LASIGE research unit since 2009. His main research interests include the dependability of distributed systems and the efficient, secure storage of large-scale critical data.

Nuno Neves received the Ph.D. degree from University of Illinois at Urbana-Champaign, Urbana, IL, USA, in 1998. He is currently a Full Professor with the Department of Computer Science, Faculty of Sciences, University of Lisbon, Portugal. He leads the Navigators Research Group and he is on the Executive Board of the LASIGE Research Unit. His main research interests include security and dependability of distributed systems. His work has been recognized on several occasions, for example, with the IBM Scientific Prize, and the William C. Carter award. He is on the Editorial Board of the International Journal of Critical Computer-Based Systems.

Alysson Bessani received the PhD degree in electrical engineering from Santa Catarina Federal University, Brazil, in 2006. He was a visiting professor at Carnegie Mellow University, in 2010 and was a visiting researcher with Microsoft Research Cambridge, in 2014. He is an associate professor in the Department of Computer Science, Faculty of Sciences, University of Lisbon, Portugal, and a member of LaSIGE research unit and the Navigators research team. His main interests are distributed algorithms, Byzantine fault tolerance, coordination, and security monitoring. More information at http://www.di.fc.ul.pt/~bessani.