BIOBANKCLOUD
Your PaaS for Biobanking

Project number: 317871
Project acronym: BIOBANKCLOUD

Project title: Scalable, Secure Storage of Biobank Data

Project website: http://www.biobankcloud.eu

Project coordinator: Jim Dowling (KTH)

Coordinator e-mail: jdowling@kth.se

WORK PACKAGE 4:
Inter-connection of Biobanks and Clouds

WP leader: Alysson Bessani
WP leader organization: FFCUL
WP leader e-mail: bessani@di.fc.ul.pt

PROJECT DELIVERABLE

D4.2
The Overbank Cloud Architecture, Protocols and Middleware

Due date: 30th November, 2014 (M24)
Deliverable version: 1.0

Editor
Alysson Bessani (FFCUL)

Contributors
Alysson Bessani, Ricardo Mendes, Vinicius Cogo, Tiago Oliveira, Nuno Neves and Ricardo Fonseca (FFCUL)

Disclaimer
This work was partially supported by the European Commission through the FP7-ICT program under project BiobankCloud, number 317871.

Contents

Document History

Version	Date	Description	Authors	Reviewers
0.1	2014-11-01	First draft.	Alysson Bessani Ricardo Mendes Vinicius Cogo Tiago Oliveira	
0.2	2014-11-27	Second draft.	Alysson Bessani Ricardo Mendes Vinicius Cogo Tiago Oliveira Nuno Neves Ricardo Fonseca	
1.0	2014-11-29	Final version.	Alysson Bessani Ricardo Mendes Vinicius Cogo Tiago Oliveira Nuno Neves Ricardo Fonseca	

Executive Summary

The data flood coming from life-science research organizations is broadly recognized as an immediate challenge that needs to be addressed. The lack of resources for timely preparing collaborative private infrastructures and, in some cases, legal constraints prohibiting public cloud usage harden this scenario. In this deliverable we present CHARON, a cloud-backed file system capable of storing and sharing big data in a secure and efficient way using multiple cloud providers and storage repositories. CHARON is secure because it does not require trust in any single entity, and it allows storing different data types in different locations to comply with required security premises. CHARON is efficient because our solution implements state-of-the-art data management techniques, such as transferring data blocks instead of whole files, prefetching sequential blocks, and uploading data in background. Our preliminary evaluation presents results from common file system benchmarks and from a new benchmark created for simulating the IO of common bioinformatics workflows. CHARON provides the underpinnings for securely storing data in public clouds and for integrating biobanks in the OVERBANK.

Chapter 1

Introduction

Chapter Authors:
Alysson Bessani (FFCUL).

The BiobankCloud FP7 project aims to develop a cloud computing and storage platform as a service (PaaS) for biobanking. Such platform will provide scalable and secure storage integrated with data-intensive tools and algorithms, and will support data sharing between biobanks using public clouds without endangering data privacy. Work package 4 is specifically assigned to the last two tasks: interconnecting individual biobank clouds, and allowing them to use public clouds. The previous WP4 deliverable (D4.1 [49]) overviewed the state of the art in the integration of biobanks and data repositories, and delineated the preliminary OVERBANK architecture and its main components.

Deliverable D4.2. This document describes the work done within the context of WP4 in the 2nd year of the BiobankCloud project. The centerpiece of the document is the description and evaluation of the CHARON cloud-backed file system, which is the basic infrastructure that will support biobank federations, i.e., the OVERBANK. Besides the description of the main aspects of the system, the deliverable also contains additional chapters detailing some innovative techniques used in the design and evaluation of CHARON.

Organization of the document. Besides this introduction, the remaining of this deliverable is divided in five chapters, namely:

- Chapter 2 describes the design, implementation and evaluation of CHARON, the cloud-backed file system providing the storage and communication underpinnings of the biobank integration. This chapter provides an overview of all the innovative techniques employed in the system, including the ones detailed in the following chapters.
- Chapter 3 surveys the access control techniques provided by several cloud storage services, and proposes a set of protocols to securely share data using different services. These protocols were implemented to enable the CHARON security model.
- Chapter 4 formalizes a Byzantine-resilient composite leasing algorithm used in CHARON and proves its correctness.

- Chapter 5 describes a novel file system macrobenchmark based on bioinformatics workflows called FS-BioBench. This benchmark emulates the I/O of representative tasks commonly executed in the NGS life-cycle, based on the interactions between bioinformatics researchers and data repositories. This benchmark was used in the evaluation of several file systems, including CHARON (as described in Chapter 2).

- Chapter 6 presents a resilient and adaptive network communication channel (RANC), which provides robust, fault-tolerant and secure communication to applications. This component will be used for implementing direct channels between CHARON clients.

3rd year. For the third year of the project, workpackage 4 will be focusing its effort in the integration of CHARON with the other BiobankCloud PaaS components, in particular with the cluster-based PaaS file system (described in D2.3), the security toolset for federated authentication and authorization (alpha version described in D3.3) and the web interface of the platform. Finally, we are following the progress of WPs 2, 3, 5, and 6 to contribute to a better integration of all BiobankCloud solutions in the PaaS.

Chapter 2

CHARON: A Dependable Biobank Data Sharing and Storage Infrastructure using the Cloud-of-Clouds

Chapter Authors:
Ricardo Mendes, Tiago Oliveira, Vinicius Cogo, Nuno Neves and Alysson Bessani (FFCUL).

2.1 Introduction

Cost-effectiveness, infinite scalability and ease of use are some of the reasons for the popularity of cloud storage services both for personal [63] and institutional [106, 48] usage. In particular, the (almost) infinite scalability of these services make them natural candidates for dealing with the data flood coming from life-sciences research institutions, in particular due to the cost-effectiveness of next generation sequencing technology [44, 68, 86].

Unfortunately, several of these institutions are still reticent in adopting public cloud services for three main reasons. First, few bioinformatics tools and systems are already integrated with public clouds, which difficult the use of clouds by researchers – who in many cases are not systems specialists. Second, as with most organizations dealing with critical datasets, there are concerns about trusting their important data to externally-controlled services that are occasionally subject to unavailability and security incidents [62, 81]. Finally, depending on the nature of the data being manipulated, there are legal restrictions impeding such institutions to outsource the storage and manipulation of the datasets [53].

These concerns are specially important for biobanks [75, 94]. Such institutions were originally designed to store biological samples that can be later retrieved for research purposes. More recently, such institutions are becoming responsible also for storing the data and the analysis of such samples, in what is being called e-biobanking [105]. The fact of these organizations lack the infrastructure for storing and managing such potentially huge amount of data makes biobanks a perfect user for cloud (storage) services. Furthermore, the cloud computing model facilitates the integration and sharing of data among biobanks, serving as a managed repository for public and access-controlled datasets, thus enabling research initiatives that are not possible today due to the lack of sufficient number of samples in a single institution [54].

Motivated by these problems, we designed CHARON, a cloud-backed file system capable of storing and sharing big data in a secure and efficient way with minimal management and no dedicated server infrastructure. CHARON builds upon recent work on multi-cloud (or cloud-of-

clouds) data replication [36, 46, 47] to avoid having any single cloud service as a single point of failure, using instead distributed trust for operating correctly even if a fraction of the providers are unavailable or misbehave. To comply with data protection legislation, CHARON allows storing datasets in different locations (cloud-of-clouds, single cloud or client-located private repository). All these locations provide different levels of guarantees, according with client's requirements, without disallowing the sharing of such datasets with standard command line calls such as POSIX' `setfacl`. In the case of data stored in private repositories, the sharing is sometimes allowed when the parties are in the same country.

CHARON exploits a *serverless design* in which clients can maintain their private secure repositories to store critical data. All the other data is stored either in a single cloud storage or in a cloud-of-clouds, requiring no dedicated server, i.e., it does not depend on any server deployed in any IaaS. This requires a set of Byzantine-resilient data-centric algorithms [35, 47], including a novel leasing protocol for avoiding write/write conflicts on files. Furthermore, the system is capable of handling big data by dividing them in blocks (for better interact with cloud services), employing erasure codes (for storage-efficiency), using prefetching (for accelerating downloads) and background uploads (for better usability). These characteristics make CHARON substantially different from previous distributed or cloud-backed file systems [37, 77, 87, 106, 48] (see Section 2.7).

In summary, this chapter presents the following contributions:

- The design and implementation of CHARON, a new cloud-backed file system designed for facilitating data sharing and storage targeting the needs of biobanks and other life-sciences research institutions;
- A fault-tolerant data-centric lease algorithm that exploits different cloud services currently available for implementing multi-cloud locking;
- A new benchmark that simulates the I/O of representative workflows commonly executed in biobank data sharing and processing.

2.2 Life Science Data Sharing and Storage

Although CHARON is a generic cloud-backed file system, its design was motivated by the need to support controlled data sharing and archival among life-science research centers, eventually distributed over large geographical areas, with minimal investments in infrastructure.

Sharing bioinfomatics data. Nowadays, most bioinformatics data sharing is based on public repositories [100]. As expected this solution has important limitations, since relevant parts of the information cannot be disseminated to everyone (e.g., human genomes [94]), as there is the need to impose access constraints [105]. At least three levels of access have to be uphold, namely private (data is only visible locally), protected (authorized partners may see the data) and public. Recently, some efforts were made to maintain and share a set of minimal metadata information about biobank studies [92, 110], but very little support exists on how to effectively carry out these tasks.

In US there have been a few proposals to create large data warehouses for concentrating the storage and processing of genomic sequence information [68]. In such scenario, the main engineering concern is to make the warehouse infrastructure scale to large data flows, something

that can often be addressed with techniques equivalent to the ones employed in commercial Internet-scale services [45].

Unfortunately, this model cannot be applied in many other regions of the globe (e.g., Europe), due to the required decentralization in management and country-specific legislation about personal data [53]. In fact, even US research centers will also have to face these issues, for instance if they want to collaborate with foreigner counterparts or if individual states decide to pass their own laws about genomic data manipulation.

CHARON tackles these problems by allowing institutions to share and archive data about organism samples and studies in a flexible, legally-compliant and secure cloud-based environment. This will enable research programs that otherwise would be hard or even impossible to establish. For example, 20–50k samples are required to study the interactions between genes, environment and lifestyle that enables or inhibits a complex disease [108]. The rarer the disease is, the longer one takes to gather all necessary samples [54]. Given the rarity of some diseases, it is extremely unlike that a single hospital or research institute will have the required number of samples.

Benefits and limitations of public clouds usage. Life-sciences research institutions and biobanks are, in principle, perfect clients of public clouds. First, the core business of these organizations is not to maintain a huge (and costly) data storage and processing facilities, and thus they would be natural candidates to exploit the ease-of-use of community or public clouds [98]. This is attested, for instance, by the large cloud adoption in bioinformatics in recent years [60]. Second, there is a flood of data currently being generated by Next Generation Sequencing (NGS) machines [44, 78, 86, 98] that needs to be stored, processed and archived by such institutions. This demand calls for an almost infinite scalable storage, which can be easily supplied by public cloud services at very competitive prices. Third, much of the utility of biobanks is related to their capability of sharing samples of data and studies. Consequently, it should be possible to make these studies and data available to other partners in a federated environment. Finally, a vast amount of money was invested in the creation of collaborative e-science infrastructures, but ultimately all these services are a burden to maintain, and eventually are discontinued. Cloud services would reduce substantially the maintenance effort, simplifying and extending the period of utilization of these infrastructures.

Such perfect matching has been leading to an impressive growth in platforms that provide an ecosystem of bioinformatics applications and third-party tools (including storage). Notable examples are the Illumina's BaseSpace [6] and Galaxy [28], both built on top of Amazon Web Services (AWS). Regrettably, these platforms follow a model quite similar to the data warehouse described previously, in which all processing is done inside a closed system (in this case, the AWS data centers). As discussed previously, legal constraints and industry secrecy will always make it difficult for ubiquitous use of such platforms.

In this chapter, we describe a solution that supports data sharing among federated parties, and efficient and durable archival of big data without requiring any kind of server maintenance or trust in a single managing entity. Furthermore, this solution enables flexible configuration under acceptable costs, providing a level of dependability proportional to the criticality of the stored data.

2.3 CHARON Overview

CHARON is implemented as a user-space file system that can be mounted on client machines, providing a near-POSIX interface to access an ecosystem of multiple cloud storage services or even to transfer data between clients. As in many other distributed file systems, it separates file data and metadata in different objects that are stored and managed in diverse ways. By file data we mean the content of each file (i.e., the user information saved, also called in this document file object), while the metadata is the set of attributes associated with it (i.e., file name, location, parent directory, time of creation, permissions, etc). Furthermore, the system can keep the file contents at various locations, according to the file owner preferences.

Figure 2.1 illustrates such diverse policies in action. In the example, the namespace tree has six nodes: directories `d1` and `d2`, and files `A`, `B`, `C` and `D`. CHARON maintains the namespace tree, together with the files' metadata, replicated in multiple cloud providers, forming a cloud-of-clouds storage. The rationale for this decision is to keep the file system structure secure and available by exploiting (1) the extremely high availability of cloud-of-clouds (which is expected to be greater than any local infrastructure or individual cloud) and (2) the secure protocols employed in this type of storage [47, 46]. The idea is to maintain only soft state in CHARON clients (the Biobanks in the example), which can be reconstructed after a crash by fetching data from other sources, and store all information about the system organization on the clouds.

Although the metadata is entirely kept in a set of public clouds, file contents may need to be stored in distinct locations to comply with privacy and criticality requirements (e.g., affecting sequencing and clinical data). This is valid even if clouds were completely secure (see Section 2.5.1). The reason why private data cannot be stored in such environment is not technical but legal, e.g., the current European legislation prohibits certain genomic-related information to be placed in countries that do not comply with specific restrictions [94, 53]. Therefore, CHARON must be able to store file contents transparently in diverse locations to operate under these constraints.

Figure 2.1 shows that file `B` is saved in the cloud-of-clouds (whether it is or not shared), while file `A` is stored locally because it cannot leave Biobank 2. File `C` is private but can be shared between Biobanks 1 and 2, thus being stored in the associated two sites (but not in the cloud-of-clouds). This scenario is useful, for instance, with datasets that have to comply with local regulations, saying that they can only be shared within the same country. The sharing of these private files has however to be carried out over a secure channel (see Chapter 6). Finally, CHARON also supports the storage of not-so-critical and non-privacy-sensitive files in a single public cloud, as it is done with file `D`. This is interesting for the cases in which decreasing costs is paramount, since storing in a cloud-of-clouds can cost between 50-100% more than in a single cloud [47].

CHARON implements only eventual consistency because file updates are uploaded to the cloud in background. Genomic files can often reach 300GBs, making it impossible to transmit them in a short interval of time with current networks. Another fundamental design decision was to avoid write-write conflicts and, consequently, optimistic mechanisms that rely on users/applications for conflict resolution [76, 87]. This decision is justified by the fact we are dealing with big files and non-specialist users. More specifically, (1) solving conflicts manually in big files can be hard and time consuming, specially for genetic data; (2) the users are likely to be non-experts in distributed computing and may not be aware of how to resolve such conflicts; and (3) the cost of maintaining duplicate copies of big files may be significant. In CHARON, when two clients try to open the same file for writing, at most one of them will succeed and make progress. An important contribution of our work is how to implement this

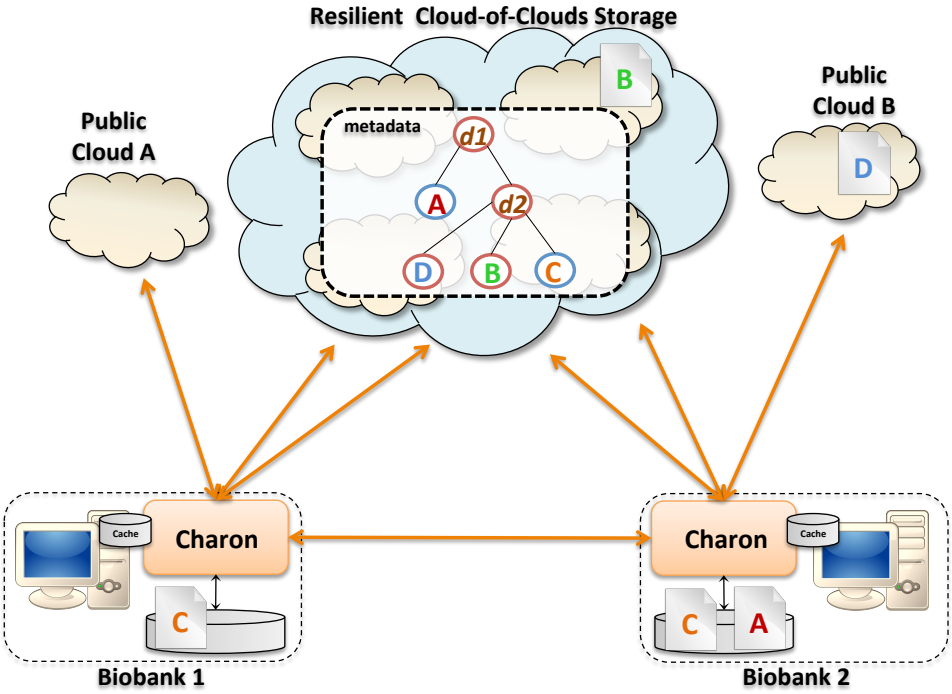


Figure 2.1: CHARON architecture.

locking algorithm without trusting cloud providers or running a dedicated server anywhere.

2.4 Leasing in the Cloud-of-Clouds

Leases are used to control concurrent accesses to objects (or resources) while avoiding version conflicts [66]. This section describes a novel leasing algorithm designed for CHARON. All protocols described in this section are formalized and proved correct in the Chapter 4 of this document.

2.4.1 Model and Guarantees

We use the same system model of traditional data-centric Byzantine fault-tolerant algorithms (e.g., [35, 47]). An unbounded number of clients can access a set of base objects (cloud services) that implement the leasing service. These clients and up to f objects can be subject to arbitrary (or Byzantine) failures. A client can initiate multiple concurrent operation invocations on the same base object, but they are executed in FIFO order. The base objects implement some form of access control to guarantee that only the clients with the necessary permissions can invoke operations (see Chapter 3). Since the notion of lease implies timing guarantees, an upper bound is assumed for the message transmission time among clients and base objects. However, *this is required only to ensure the liveness of the protocol* as safety is always preserved.

Different from previous works on leases [57, 66], our guarantees are more similar to the definition of the classical (always safe) mutual exclusion problem, in which there can be never more than one process accessing the critical section of its code (i.e., accessing a resource) [42]. We extend this by introducing the validity of a lease, or *lease term* [66]. More formally, the lease service for protecting the access to a resource is invoked through operations: *lease(T)*

to acquire a lease for a term of T time units; *renew(T)* to renew a lease for T time units; and *release()* to end the lease. These operations need to satisfy the following properties:

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

The above properties do not preclude a Byzantine client from acquiring a lease and keep renewing it constantly, neither constraint a direct access to the resource (e.g., writing a file) without having the corresponding lease. This is no significant limitation since a Byzantine client can damage all resources (files) for which it has the necessary access permissions. In the same way, a lease acquired by a client that crashes will be available to other clients after at most T time units (Time-boundedness property).

The obstruction-freedom property is the weakest liveness condition for a leasing/mutual exclusion protocol. We chose to satisfy this condition in CHARON because stronger guarantees typically require a greater number of accesses to shared objects and write/write conflicts are expected to occur rarely in practical deployments (i.e., created files are rarely updated).

2.4.2 Byzantine-resilient Composite Lease

Contrary to previous approaches for implementing Byzantine-resilient data-centric mutual exclusion [35, 47], which directly rely on quorums of minimal storage services to establish lease acquisition, we propose an approach in which the lease is defined in only two steps. First, *non-fault-tolerant base lease objects* are built using a single service from a specific cloud provider, and then a set of $n = 3f+1$ of these lease objects are composed to implement a *f-fault-tolerant lease object*. Besides its obvious modularity, this method enables the creation of more efficient lease objects that rely on evolved services (e.g., queues, test-and-set-enabled databases, or transactions) made available by the diverse cloud providers. Furthermore, it allows us to address the heterogeneity of these services, for instance, with regard to guarantees and exported operations.

The key idea of such *composite lease* construction, as specified in Algorithm 1, is to make a client succeed in obtaining the lease if it manages to get leases from a quorum of $n - f$ out-of n base lease objects implemented over distinct cloud providers. In the algorithm, as well as in the following sections, we did not discuss the algorithm for renewing a lease since it is similar to the lease algorithm with an additional cloud access for garbage collection (i.e., removing the information related with the old lease).

In order to acquire a composite lease, a client calls in parallel the lease operation in each base lease instance (Lines 5-6). After that, it waits for $n - f$ success or $f + 1$ failure responses (Line 7). In the first case, the lease is acquired and the operation can return. Otherwise, the client needs to release all the leases that were potentially obtained — the granted leases and the leases for which no response has arrived (Lines 11-12). Then, it backoffs and tries again after some time (Line 13). This procedure is repeated until $n - f$ success responses are obtained or a certain time has elapsed (not showed in the algorithm for legibility reasons). To release a lease, a client could simply wait for it to expire. However, to allow other clients to proceed sooner, it invokes in parallel the $release$ operation in all base lease objects (Lines 16-19).

ALGORITHM 1: Composite resource leasing by client c.

```
1  function lease(time) begin
2      result ← false;
3      repeat
4          L[0 .. n − 1] ←− ⊥;
5          parallel for 0 ≤ i ≤ n − 1 do
6              L[i] ←− baseLease_i.lease(time);
7          wait until i : (|{L[i] = true}| ≥ n − f) ∨ (|{L[i] = false}| > f);
8          if |{i : L[i] = true}| ≥ n − f then
9              result ← true;
10         else
11             for i : (L[i] =⊥) ∨ (L[i] = true) do
12                 baseLease_i.release();
13             sleep for some time;
14     until result ≠ false;
15     return result;
16 function release()
17 begin
18     parallel for 0 ≤ i ≤ n − 1 do
19         baseLease_i.release();
```

2.4.3 Base Lease Implementations

Base lease objects can be constructed in various ways, for example by resorting to cloud services that have better performance or that support functions with stronger synchronization power [69]. In the next subsections, we describe four different base lease objects algorithms that were developed with services currently available at popular cloud providers.

Although employing different cloud services, all base lease objects were designed following a group of similar techniques, which allow us to deal with malicious behavior and manage the duration of a lease. First, the lease operation requires the successful creation of (various flavors of) *lease entries* in the cloud service. Second, clients have to garbage-collect the outdated (or invalid) lease entries that they create to avoid wasting resources. In most implementations, this cleanup requires at least one cloud access in the lease and renew algorithms. Third, all algorithms sign the lease entries before writing them to the cloud, to ensure that no cloud can create or corrupt leases. Fourth, malicious clients can only attempt to corrupt the leases of resources for which they have the necessary permissions, and therefore they can only hurt themselves (or eventually partners that mistakenly gave them access to the resource). Lastly, clients do not add a timestamp to the lease (to mark its starting period), instead they rely on the cloud service to add a timestamp to the created entries. In the same way, clients do not rely in their clocks to check the period of validity of a lease, but instead get from the cloud the current time (which is automatically returned in every operation).

Storage Services

Object storage is one of the most common and popular service made available by cloud providers. We designed an algorithm that implements base lease objects in the same line as the one developed for DepSky [47], but without relying on synchronized clocks among clients and adapted to a single cloud provider (instead of a cloud-of-clouds). The algorithm works in services like Amazon S3 [3], Google Storage [13], Azure Blob Storage [56] and Rackspace Files [23], because they all have a key-value store interface [46] with strong consistency guarantees for functions related to file/entry creation.

In our algorithm, to obtain a lease, the client first lists a lease container associated with the resource. If no valid lease entries are found, it inserts a new entry in the container. To complete the operation, the client lists again the container to verify if other lease entries were inserted concurrently. If there is contention and the client finds another valid lease entry in any of the two listing, it removes its own entry (if already inserted) and returns $false$. Otherwise, the lease acquisition succeeds.

The release operation corresponds only to the removal of the lease entry.

Augmented Queues

Several cloud providers have services implementing a queue for the coordination of processes. Some of these services offer strongly-consistent enqueue/dequeue and list functions, thus offering an augmented queue shared memory object [69]. In the following, we describe an algorithm for base lease objects build on top of Windows Azure Queue [20] and RackSpace Queue [19].

A client acquires a lease when its lease entry is the first valid one in the queue. The first step of the lease operation is to list the queue to see if there are other contenders. If the queue is empty the client enqueues a signed lease entry. Then, it lists the queue again to check that its entry is the valid one with the lowest index. If this is the case, the client obtains the lease.

If the queue is not empty after the first list, the client verifies if some entry in the queue is valid and not created by itself, in which case the lease acquisition fails. Otherwise, if the lease entry belongs to the client (e.g, is a renew operation), the client pushes a new lease entry to the queue. Additionally, it removes the entries that were observed in order for its lease to be at the head of the queue.

To carry out a release operation, the client only tests if it stills holding the lease. It then removes the entry in case this was true.

Amazon DynamoDB

Amazon DynamoDB [2] is a NoSQL database-as-a-service that can be utilized both by internal (i.e., VMs deployed in Amazon EC2) and external clients. Contrary to its initial versions [61], used in the Amazon web store, DynamoDB provides strongly-consistent storage and a test-and-set function that enables the implementation of very efficient base lease objects.

The first step to acquire a lease is to search for a lease entry for the specific resource in the database. Whenever such lease entry exists, it is checked the ownership, validity time and integrity. If that lease belongs to another client and is still valid, the operation returns $false$. Otherwise, the client creates a new lease entry and uses the database test-and-set function ($testAndSetItem$) to put the entry. The function ensures that the new lease entry is only set if in the meanwhile no other client added an equivalent entry, returning $true/false$ in case of success / failure.

The release operation consists basically in the removal of the lease entry from the database.

Interestingly, DynamoDB is the only service among the ones we tried that is charged monthly, instead of for each function execution. The charging model depends on the service throughput required from the service, in terms of reads and writes per second. For instance, each 10 writes/sec or 50 reads/sec of throughput costs $0,00735$\$/hour [2].

Google Datastore

Google Datastore [10] is a database service that can be accessed through the internet. Differently from Amazon DynamoDB, it does not have a test-and-set function but supports atomic transactions, which allow the implementation of obstruction-free base lease objects.

The lease operation is executed between the boundaries of a transaction. The client first performs a lookup for a lease entry in the database service. If there is a valid lease entry, the transaction is aborted and the operation fails by returning $false$. Otherwise, the client inserts its own lease entry and attempts to commit the result. Committed transactions indicate that there were no conflicts and no other lease entries were inserted, and thus the lease operation can return $true$. In the opposite case, the lease acquisition fails with a $false$ result.

To implement release, the client opens a transaction and executes a lookup to check if the lease belongs the it. If this is the case, it deletes the lease entry and commits the transaction.

2.4.4 Comparing Base Lease Objects

The algorithms described in previous section can be used for implementing base lease objects in eight different cloud services. Table 2.1 summarizes the properties of the described base lease algorithms.

Service	lease/renew/release	costs (μ\$)	Progress
Amazon S3	3/4/2	15	Obst.-Free
Google Storage	3/4/2	30	Obst.-Free
Azure Blob Storage	3/4/2	0, 108	Obst.-Free
RackSpace Files	3/4/2	8, 640	Obst.-Free
Azure Queue	3/4/2	0, 15	Dead.-Free
RackSpace Queue	3/4/2	30, 144	Dead.-Free
Amazon DynamoDB	2/2/1	free*	Dead.-Free
Google Datastore	4/4/3	1, 2	Obs.-Free

Table 2.1: Base lease objects built with various cloud services. The table shows for each service: the number of cloud accesses required for executing lease/renew/release operations in absence of contention; the monetary costs of executing a $lease$ 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.

In the table it is possible to see that most base objects require three cloud accesses for implementing the lease operation. In terms of the progress property satisfied by the algorithm, it is possible to see that most of the algorithms not based on cloud storage services satisfy Deadlock-freedom [42]. It is worth to notice that, even if we use n base objects that satisfy Deadlock-freedom to build a composite lease, the later will only satisfy Obstruction-freedom.

In terms of costs, the table shows that there is a huge difference between the costs of running the lease algorithm in the cloud services of the table. However, our approach, even if used with the most expensive services, will be less costly than running a fault-tolerant lock service in cloud VMs [48].

2.5 CHARON Design and Implementation

CHARON was implemented in Java as a FUSE-based user-level file system. This section explains the most relevant features of CHARON and how they were implemented. First, it describes the various storage locations that are supported and how they can be identified. Then, it presents how the file data and metadata are organized and managed. Finally, we provide an example of how system calls are handled in our system and define the security model implemented in CHARON.

2.5.1 Storage Locations and Namespace

CHARON stores metadata information in a cloud-of-clouds, but gives complete flexibility to the user to decide where to store file contents. Three different locations are supported in our current implementation: a cloud-of-clouds, a single (public) storage cloud and a private repository (e.g., a private cloud). These alternatives are able to address all placement requirements we have encountered with life sciences applications. More specifically, the cloud-of-clouds can be used to store critical data that need the availability and confidentiality provided by the multi-cloud scenario, a single storage cloud could save, for example, studies that can be public for everyone, and finally, private repositories must be used to keep, for example, clinical data from human samples that cannot leave the boundaries of a particular biobank [94].

The cloud-of-clouds location is the default option for CHARON, but users may select any of the other two possibilities to indicate where to put the file objects. The location is defined using the semantic cues introduced in the WheelFS [101], in which the pathname specifies the location of a particular object. Specifically, in CHARON, users are only able to use cues over folders, i.e., they can specify the location of each folder at the time of its creation. Henceforth, each file created in a folder with a location different from the default one, will be stored in that defined location.

In concrete, users must use the cue `.Location` to determine where a directory should be created. For example, the path `/share/.Location=S3/data` informs that the directory `data` is in Amazon S3 (in a pre-configured account). On the other hand, the directory `private` of path `/share/.Location=myrepo:23456/private` will be located in a private repository called `myrepo`, by connecting to port `23456`.

2.5.2 Metadata Organization

Metadata is the set of attributes assigned to a file/directory, including the name, location of the data, parent directory, time of creation, permissions, among others. Independently of the location of the data blocks (e.g., a private repository), CHARON stores all metadata in the cloud-of-clouds. The rationale for this decision comes from the accessibility and availability guarantees achieved with this configuration, which are potentially higher than the other alternatives [47]. Furthermore, privacy is not endangered because metadata is encrypted before being stored, and legal concerns only apply to the content of the files.

Namespaces management. All the metadata is stored within namespace objects, which encapsulates the hierarchical structure of the files and directories in a subdirectory tree. There are two kinds of namespaces: *personal namespaces* (PNS) and *shared namespaces* (SNS). A PNS stores the metadata of 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 PNS associated with itself. On the other

hand, each shared folder is associated to a SNS. In this way, a client has access to as many SNSs as the shared folders it can access. All references to these SNSs are stored in the client PNS.

Figure 2.2 illustrates a scenario where a client A shares a folder with another client B. In the figure, the shared folder is assigned to the SNS1, while the PNS1 and PNS2 are associated with clients A and B, respectively. The figure also illustrates how a set of files relate with these namespaces. Files A and B are private to their owner (clients A and B, respectively) while file C, which has two blocks, is shared among the two clients. As can be seen in the figure, since files A and B are private, the reference to their content is maintained in their owner PNSs. In the case of file C, the reference to the blocks is stored in SNS1.

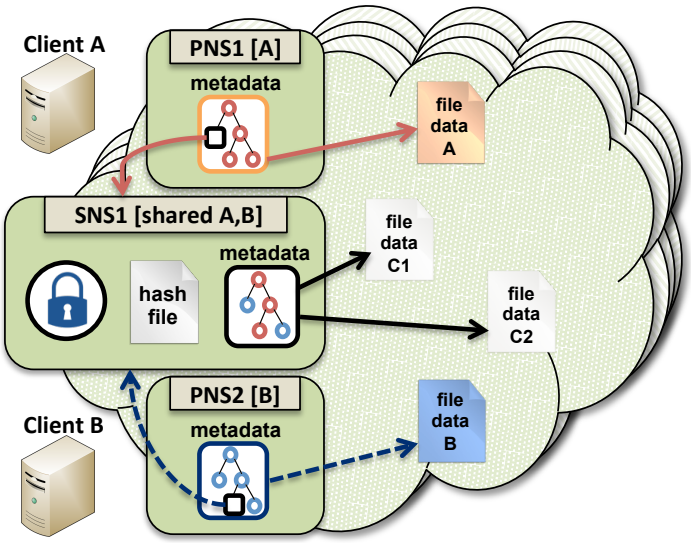


Figure 2.2: Objects maintained in the cloud by CHARON.

Private and shared namespaces differ mainly in the way the hashes of the most recent versions of the files' data blocks are stored. In PNSs this information is stored together with the remaining files' metadata. When the PNS needs to be stored, these hashes are serialized together with the other metadata for the files and sent to the clouds. On the other hand, in an SNS these hashes are stored in a separate file (see Figure 2.2). The rationale for this decision will be discussed bellow.

Another important difference between PNSs and SNSs is that the latter is associated with a lease, that must be acquired before any update is executed on a file or directory associated with the namespace.

Dealing with shared files. The PNS' metadata is downloaded from the cloud-of-clouds only once, when the file system is mounted. In turn, SNSs' metadata is periodically retrieved from the clouds to find new versions of it (versions updated by other user). The reason for having a separated hash file in SNSs comes from this need. Since a SNS carry all metadata of all files inside it, having the hashes of all blocks of all files together with this info could significantly increase the monetary and performance costs of periodically downloading the shared metadata. Given this, the hash file is only refreshed when a file is opened (either for read or write). Moreover, storing the hashes of all data blocks in a single hash file can be unpractical due to fact that the size of such file could grow linearly with the number of files stored in the SNS, making it very costly to fetch the hash file of highly populated SNSs. To circumvent this problem,

CHARON defines a maximum number of entries allowed in each hash file (currently we use 100 hashes). When this number is reached, newer hashes are saved in a separated file. There is however an exception to this rule: the hashes of data blocks from the same file are always kept in one hash file.The reason for this exception is to avoid fetching several hash files when opening a large file. SNSs are responsible to store the location of the correct hash file of all files they maintain.

In order to avoid write/write conflicts, the user must obtain a lease over the entire SNS before updating a shared file. Any modification performed by the client in that SNS is asynchronously propagated to the location where the file is located (e.g., the cloud). After such upload completes, the corresponding metadata (i.e., the SNS hosting the file) is updated in the clouds. The lease is only released after the metadata is uploaded to the cloud-of-clouds. Such background propagation is crucial for the usability of the file system, since waiting for such cloud synchronization incurs a high latency. However, other clients may still perform read-only operations on directories and files belonging to the leased SNS. Concretely, they will be able to read data that corresponds to the latest metadata version uploaded.

2.5.3 Data Management

This section presents the most relevant techniques used by CHARON to improve its efficiency on file data management. It includes a multi-level cache system, transference of data blocks instead of whole files and prefetching sequential blocks, among others.

Multi-level cache. CHARON uses the local disk to cache the most recent files used by clients. Moreover, it also keeps a fixed and small main memory cache to improve subsequent data accesses over open files. Both the main memory and disk caches implement LRU (Least Recently Used) policies. The use of cache not only improves performance, but also decrease the operational cost of the system. This happens because cloud providers charge the download of data, but usually does not charge data upload (as an incentive to store data on their facilities [5, 14, 24, 33]), which means that the cost of operating CHARON corresponds roughly to the cost of keeping the data in storage plus the download of new versions of files.

Working with data blocks. Managing large files in cloud-backed file systems brings two main complexities. First, reading (resp. writing) whole files (as is done in SCFS [48]) from (resp. to) the cloud is impractical due to the huge time required for downloading (resp. uploading) such files. Second, cloud-backed file systems strongly rely on cache for ensuring usable performance [48, 106], however sometimes these files are so big that they do not fit the allocated cache. This last problem is specially frequent when considering the main memory cache.

CHARON addresses these issues by splitting (big) files into fixed-sized blocks. The block size is configurable, but we currently use 16MB since this size offers an interesting balance between latency and throughput [47, 48]. This strategy is similar to what is done in other cloud-backed file systems, such as BlueSky [106] that uses data blocks of 4MB length.

A block with few megabytes is relatively fast to load from disc to memory, can be transferred from/to clouds in a reasonable time, and is still small enough to be maintained in main memory. This last advantage is extremely important to absorb bursts of sequential accesses, for example, reading a 16MB file entails 4096 subsequent accesses since Linux fetches blocks of 4kB by default. Additionally, our approach is also cost-effective because, in case a cached file being updated, only the modified data blocks need to be transferred to its storage location.

In CHARON, each cached data block has its integrity validated by using a hash that is stored in the cloud-of-clouds, together with the metadata. If we fetch the hash of a block that does not match the cached block, the system becomes aware that a new block version is available. This could happen, for instance, when a shared file is modified by another client or, similarly, with a private file if there are two CHARON instances mounted with the same credentials. In the most common scenario, private files are expected to be maintained in the local disk cache, which provides a performance similar to a local file system.

Prefetching. Scenarios performing sequential reads motivate this optimization, which makes the system automatically starts fetching subsequent data blocks even before they are requested by the client. This is implemented through a background thread pool responsible for prefetching data blocks from any location as soon as a sequential read program is identified. The heuristic verifies when half of a block is read sequentially, and then triggers the prefetching mode. If in the meanwhile, a file being prefetched is closed, all enqueued requests for that file are removed from the prefetching queue. This strategy is important to improve the performance of several bioinformatics workflows since most of them read files sequentially, as can be observed in Section 2.6.3 (and Chapter 5).

Supporting diverse storage locations. The use of a cloud-of-clouds to store data brings benefits in terms of resilience to failures, but also avoids vendor lock-in problems. CHARON resorts to DepSky [47] to ensure that files are stored considering availability, integrity and confidentiality properties, even if a fraction of providers become faulty. High availability is obtained by storing the data in a set of public clouds instead of in a single entity (as some other cloud-backed storage systems do [26, 27, 99, 106]). It uses data-centric Byzantine quorum protocols [85] that require a set of $n = 3f + 1$ cloud services, with at most f of them are subject to Byzantine faults. DepSky uses storage-optimal erasure codes to store only portions of a file on each cloud, making the storage of a file in a cloud-of-clouds at most twice more costly than storing it on a single cloud for $f = 1$. Integrity is provided by maintaining the validity of the stored data using a cryptographic hash. Confidentially is obtained by encrypting the data before storing it, and distributing key shares to the cloud providers in such a way that a single provider will not be able to recover the whole file.

However, the inherent redundancy and security of a single cloud provider may be enough for not-so-critical files. Therefore, this option can be used to reduce the costs involved with maintaining the file objects.

The private repository is employed to store files containing privacy-sensitive information. The system uses secure data transfer protocols (e.g., SFTP or RANC, see Chapter 6) to directly transfer data between private repositories, and to provide data to a trusted data client.

All the three locations are configurable. For the cloud-of-clouds scenario we currently support Amazon S3 [3], Google Storage [13], Windows Azure [56], Rackspace Cloud Files [23] and Hp Public Cloud [15], from where each user can choose a different configuration. For a single cloud, CHARON can employ any of the cloud storage providers enumerated above. The private repository can be another disk in the same machine, a dedicated private storage infrastructure, or an external trusted facility (e.g., a governmental repository [39]).

2.5.4 System Operation

Figure 2.3 illustrates the execution of common POSIX system calls such as open, write, read and close in our system. Specifically, this figure considers a scenario where a client opens a file to perform a write or a read on a non-cached data block. In the Figure 2.3 it is possible to see six different components of the system: (1) the `Lease Worker`, which is responsible to execute the lease protocols described in Section 2.4; (2) the `Metadata Cache`, which is where the metadata is cached; (3) the `Memory Cache`, which represents the main memory data cache; (4) the `Disk`, which is used to cache the most recently accessed file blocks; (5) the `Cloud-of-Clouds Storage`, and (6) a background thread, which is responsible for maintain the `Metadata Cache` updated (as described in Section 2.5.2). In the following we describe the execution flow for each system call.

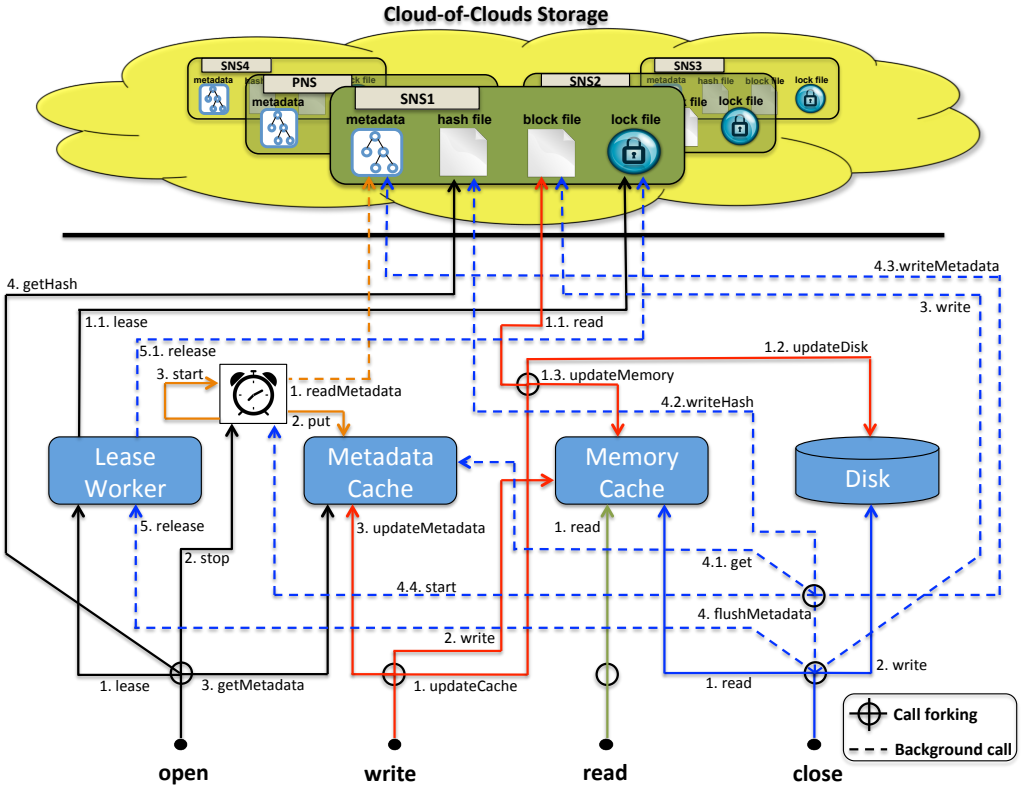


Figure 2.3: Common file system operations in CHARON. The following conventions are used: 1) at each call forking (the dots between arrows), the numbers indicate the order of execution of the operations; 2) each file system operation (e.g., open/close) has a different line color.

Open. To avoid write/write conflicts, when a file is being opened for writing, the first task to be executed is to call the `Lease Worker` to obtain the lease over the SNS hosting the file. If the lease is not available, the call returns an error to the application and the file is not opened.

The next step is to stop the background thread. We do that to prevent the system of unnecessary downloads, since no one else will be able to update metadata because we have the lease for the SNS.

After that, the system gets the most recent available metadata of the file being open. In our case, the client uses the metadata from the `Metadata Cache` since it is already updated. The background thread periodically gets the metadata from the clouds and update it.

The last step of this operation is to download the hash file where the hashes of the file being opened are stored to validate the cached data blocks which will be accessed by the application.

Write and Read. For writing, the system only needs to modify the block in the `Memory Cache` and update the corresponding metadata (e.g., file size) on the `Metadata Cache`. For reading, the system only needs to retrieve the requested data from the `Memory Cache`. However, when the application is accessing a block that is either outdated or not cached, this block needs to be read from the cloud (using the hash of the most recent version) and then stored in the `Disk` and in the `Memory Cache`.

Close. The close operation has the purpose of synchronize the modified data and metadata with their respective storage locations. First, the system copies the updated data block from the `Memory Cache` to the `Disk`, and then uploads it in background to its respective location (the `Cloud-of-Clouds Storage` in the figure). When the upload ends, the system sends the updated hash file and metadata (i.e., the SNS) to the `Cloud-of-Clouds Storage`. By first saving the data and then the corresponding metadata we ensure that every time some metadata is read, the data pointed by it will be available. Next, the thread responsible for updating the metadata cache (previously stopped in the open operation) is resumed to continue updating the `Metadata Cache`. After all the data and metadata is properly saved in their target locations, the system release the lease for SNS of the file (this is usually done background, after some seconds).

2.5.5 Security Model

CHARON implements a security model where the owner of the shared resource pays for its storage and is able to define its permissions. Moreover, CHARON clients are not required to be trusted, since the access control is performed by the (untrusted) cloud providers, which enforce the access permissions for each accessed object. The CoC access control is satisfied if no more than f cloud providers misbehave. This is effective because even if an object is read from f faulty providers, no information will be obtained since all written data is encrypted using secret sharing [47].

The implementation of this model requires a mapping between the file system and cloud storage abstractions. Each CHARON user authenticates using its own credentials on each cloud provider used by the system. After that, each file or directory a user creates results in the creation of one or more objects associated with their accounts. Moreover, the system only allow sharing directories. To give permission to others for accessing a directory, i.e., to share it, an user needs to change the POSIX ACL associated with the desired directory. When this happens, the CHARON client changes the permission associated with each object in the cloud with the available service APIs (this is described in detail in Chapter 3). Concretely, to share a private directory, the system creates an SNS and give permission to the sharing users, updating also the owner PNS.

In both cases, each CHARON user id has to be mapped to the corresponding cloud accounts. Each user maintains this mapping together with its private name space in the cloud-of-clouds. Since there is no "server" to inform clients about the arrival of other clients in the system, the

discovery of new clients should be done through external means. For example, in a biobank federation, the biobanks must know the cloud identifiers from each other. CHARON provides a command line tool for managing these credentials.

2.6 Evaluation

This section presents three sets of experiments to evaluate CHARON and compare it with other local and cloud-backed storage systems. First, we discuss the latency and scalability of the base and composite leasing algorithms. Second, we present results of several microbenchmarks for evaluating the performance of CHARON in terms of metadata and data-intensive operations under different scenarios. Finally, we conclude by presenting the results of a novel benchmark, which was developed based on several significant bioinformatics workflows.

Experimental Environment. Our experimental environment is comprised by four Dell Power Edge R410 machines placed in our facilities. Each one of them is equipped with two Intel Xeon E5520 (quad-core, HT, 2.27Ghz), 32 GB of RAM, and two disks: a 146GB HDD with 15k RPM and a 120GB SSD. The operating system is an Ubuntu Server Precise Pangolin (12.04 LTS, 64-bits), running kernel 3.5.0-23-generic, Java version 1.7.0_67 (64-bits) and Python version 2.7.3.

CHARON was configured to store data in three different locations: a private repository, a single cloud provider, and a cloud-of-clouds. The cloud-of-cloud storage used in our experiments is composed by Amazon S3 (US), Windows Azure Storage (WE), Rackspace Cloud Files (UK) and Google Cloud Storage (US). For the single cloud storage, we only show results for Amazon S3, since it is the most widely used storage cloud. The private repository was located in the disk of the client machine.

For running the composite lease we also employ additional cloud services such as Azure Queue [20], RackSpace Queue [19], Amazon DynamoDB [2], Google Datastore [10].

To evaluate the performance of the file system operations (e.g., MakeDir, Write) we use the Filebench [9] tool. We compare all CHARON storage locations with a local file system (ext4) and with other cloud-backed file systems such as SCFS (non-blocking mode) [48] and S3QL [27]. Similarly to CHARON, these two systems send file updates to the cloud in background.

2.6.1 Composite Leasing

In this section we present a set of experiments to evaluate the Byzantine-resilient composite lease algorithm described in Section 2.4.2 and the non-fault-tolerant base objects from which it can be built. We focus our analysis on the lease operation, since it is the only one that will be on the critical path of any application handling shared files, as renew and release are always executed in background in CHARON.

Contention-free executions. Our composite leasing algorithm was configured in three ways: one using only storage cloud services from different providers (ST in the graphs), another using only non-storage cloud services, such as queues (NST) and a third based on the fastest base services, even if from the same providers (WE). Figure 2.4 presents the latencies of the lease and release operations for these two compositions and several base lease objects.

Regarding the individual lease objects, there are three classes of services in terms of performance: the ones operating in 200 ms (Azure Queue, DynamoDB and Azure Storage), the ones operating in 500 ms (Google Datastore, Rackspace Queue and Amazon S3) and the ones

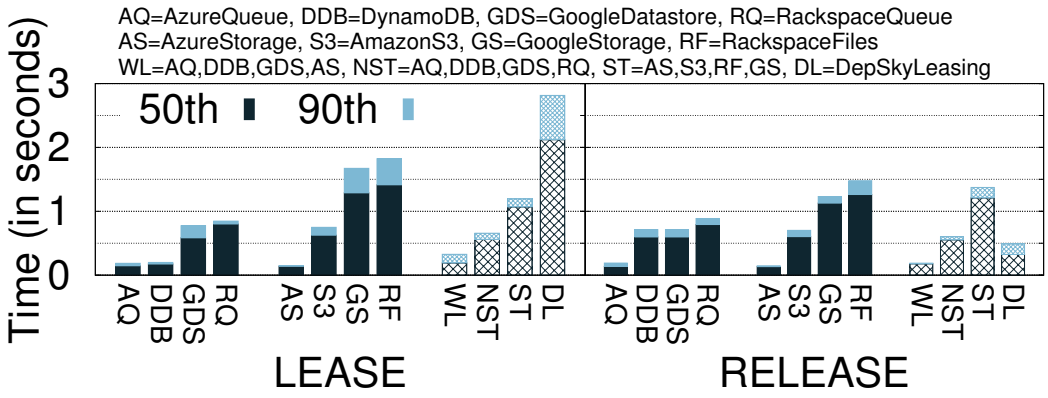


Figure 2.4: Latency of lease (without contention) for several configurations of the composite algorithm and all base services.

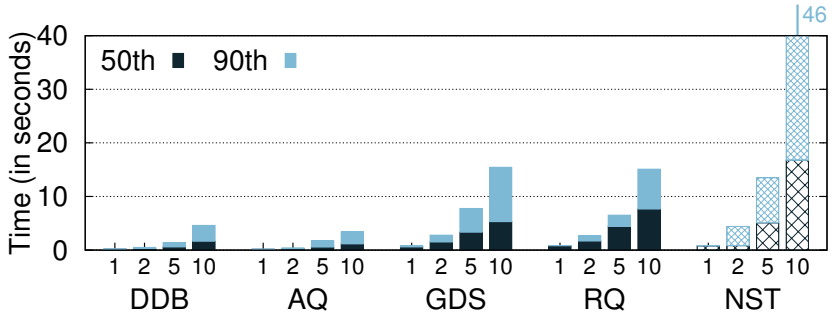
operating in more than 1 sec (Google Storage and Rackspace Files). The results for base lease objects using non-storage cloud services are in general better than the ones from storage services. We hypothesize that this happens because the storage services are throughput-oriented, and thus less efficient when dealing with small data entries.

The results for the composite lease configurations reflect the performance of their base objects. More specifically, the lease protocol waits for a quorum of $n - f = 3$ leasing acknowledgements from different services, which means that the latency of the composite lease is similar to the third fastest cloud service. For instance, the latency of NST is similar to the latency of GDS, which is worse than DDB and AQ, but better than RQ. For a pure storage-based lease (ST), we observed a lease latency 100% worse than NST. The WL configuration uses the fastest base objects available (storage and non-storage), achieving a latency 100% better than NST. The main limitation of this configuration is that it uses two services from the same provider (Azure Queue and Azure Storage), which results in less diversity and fault independence. In consequence, we reject this design and use the NST configuration in our remaining experiments. Although slower, this configuration allows the lease acquisition in around half a second, which is an acceptable time when considering the latencies of accessing a remote cloud service.

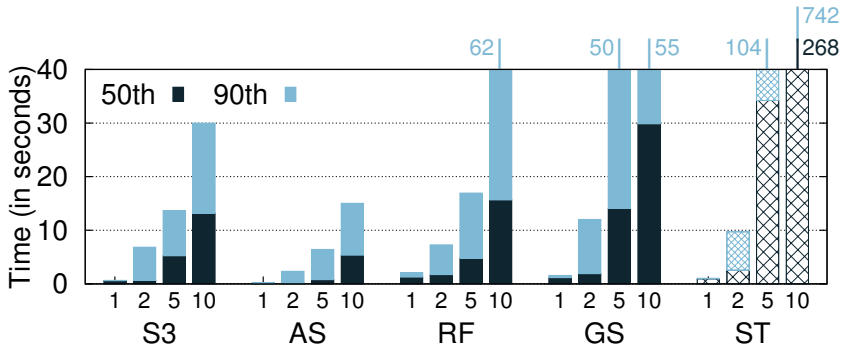
In the figure we also show the latency of acquiring a lease using the DepSky mutual exclusion algorithm (DL) [47], which is based on the same services as our ST configuration. The observed latency for DL is twice the latency of ST and four times bigger than NST (used in CHARON). This happens because the DepSky mutual exclusion access the storage clouds in phases, and not by executing base lease algorithms in parallel.

Executions under contention. Another important aspect of a lease algorithm is how the solution scales when increasing the number of clients trying to obtain leases over the same resource. We performed 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 required for a client to acquire the lease. For lease algorithms that are only obstruction-free, we use a random back-off time between 0-1 second. The results for several base lease objects and composite lease configurations are presented in Figure 2.5.

Interestingly, non-storage services (Figure 2.5(a)) provide better/faster results than storage services (Figure 2.5(b)). This happens because most non-storage services satisfy the Deadlock-freedom property (i.e., if several processes tries to get the lease, some process will be success-



(a) Non-storage services



(b) Storage services

Figure 2.5: Leasing acquire time under contention between 1, 2, 5 and 10 clients on diverse cloud services.

ful [42]), which makes them much better for dealing with contention than the storage-based algorithms, which only implement obstruction freedom (see Table 2.1).

The composite lease object (NST and ST in the figures) also provides only obstruction freedom, and thus have a super linear increase in the waiting time for getting a lease when under contention. This is expected since CHARON was not optimized for scenarios with a large number of clients trying to updated the same folder or file. The composite lease algorithm is fast with one or two contending clients but noticeably slower with 5 or (specially) 10.

2.6.2 File System Microbenchmarks

In this section we present the results of a set of microbenchmarks that evaluate different aspects of the CHARON file system.

Metadata-intensive operations. Our first experiment focus on how well the system deals with metadata intensive operations when compared with other systems. Table 2.2 presents the number of operations per second for ext4 (on SSD), S3QL [27], SCFS [48] and CHARON. We used 0-byte files to put focus on metadata management.

The results show that CHARON offers a performance within the same order of magnitude of ext4, being slower mainly due to the overhead of FUSE and its Java wrapper. Interestingly, our system was able to create directories faster than ext4. This happened because a MakeDir operation in CHARON returns when the metadata is updated in memory, while ext4 updates data structures in disk. When compared with other cloud-backed file systems, CHARON is

Operation	ext4	S3QL	SCFS	CHARON
Create	2500	45	2	500
Delete	2519	66	4	1258
Stat	22069	19474	9	15786
MakeDir	8332	4493	14	12499
DeleteDir	12498	1162	5	9998
ListDir	20927	879	6	3017

Table 2.2: Metadata-intensive microbenchmark. Values in operations/s.

faster because metadata updates are done only in memory (S3QL uses an SQLite local database for that) and later sent to the cloud in background (SCFS synchronizes every metadata operation with the cloud).

Data-intensive operations. Table 2.3 presents the results for a similar microbenchmark for the same competing systems, but now focusing in data-intensive operations. These experiments consider files of 64MB.

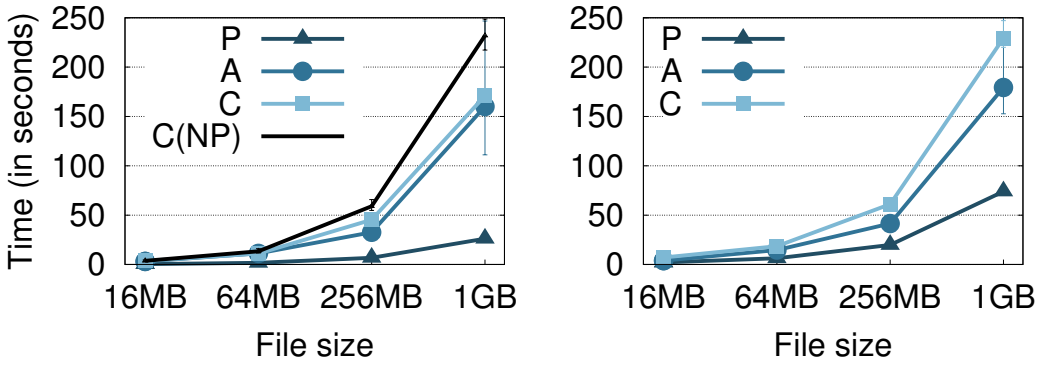
Operation	ext4	S3QL	SCFS	CHARON
seqRead	210	208	197	194
randomRead	203	204	195	195
seqWrite	62	10	16	31
randomWrite	44	5	38	35

Table 2.3: Data-intensive microbenchmark. Values in MB/s.

Our results show that ext4 and S3QL provide a similar read throughput, both for sequential and random workloads. SCFS and CHARON offers a lower, but competitive, performance for read workloads. When considering write-throughput, the ext4 local file system presents the best performance both for sequential and random workloads. However, CHARON presents $2\times$ better write-throughput (seqWrite and randomWrite) when compared with the other cloud-backed file systems, with the exception of the SCFS' randomWrite, which is slightly better than CHARON.

Read and write of big files. Efficiency in reading and writing large files is one of the main objectives of CHARON. Figure 2.6 shows the results from sequential read and write operations performed over large files (from 16MB to 1GB). We performed these experiments considering different data locations for CHARON, namely: private repository in the same network (P), single cloud in Amazon S3 (A), and the cloud-of-clouds (C). In the last scenario, the read operation is performed with and without prefetching (C-NP) to verify the impact of this optimization in the read latency. In all cases (with the exception of C-NP) CHARON uses four threads sending and receiving data from the referred locations.

As expected, reading and writing from a private repository presents the best latency, since the target location is inside our local network. The difference between the latency of using Amazon S3 or the cloud-of-clouds is quite small, especially for reading. For writing, the additional latency comes from the fact that we need to update (with half of the file) a quorum of clouds (three out of four) to finish the write, and the end-to-end latency will be dictated by the 3rd fastest cloud. Finally, the results showed that prefetching file blocks significantly improve



(a) Read (b) Write

Figure 2.6: Read and write latency for different file sizes in different locations: private repository in the same LAN (P), Amazon S3 (A), the cloud-of-clouds (C) and the cloud-of-clouds without prefetching (C-NP).

sequential reads of big files. As shown in the graph, downloading a 1GB file from the clouds using this technique decreases the whole-file read latency by approximately 23%.

File sharing latency. Our final set of microbenchmarks aims to compare the latency of sharing a file in CHARON and in other cloud-backed systems such as SCFS and Dropbox. In this way, we repeated the sharing experiment introduced in the SCFS paper [48]. The experiment consists in a client writing a file in a shared folder while another client, that also has access to this folder, tries to read the written file. Thus, we measure the time it takes from the instant a client writes a file and close it (at this time the data are only stored in disk once all the evaluated systems start uploading the data in background) until the moment the other client reads the entire file. Figure 2.7 presents the sharing results for desktop-size files (256kB to 16MB), as used in [48].

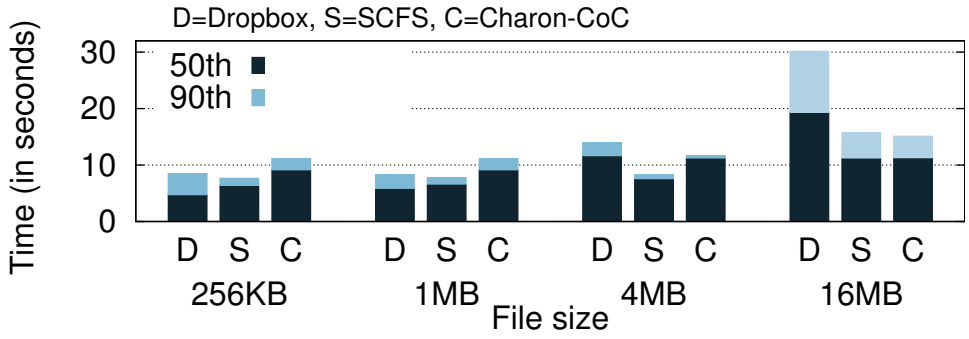


Figure 2.7: Sharing latency for different cloud-backed systems and file sizes.

Dropbox is fast for sharing small files, which is also true for SCFS. However, the sharing latency increases by a factor of four between 256kB to 16MB in these systems, while such value increases less than 20% for CHARON. This happens because CHARON fetches the hash of shared files with a periodicity of five seconds (in our experiment), which means that, there will be a delay of up to five seconds in discovering the update, independently on the size of the shared file.

2.6.3 Bioinformatics Workflows

Our previous experiments employed general file system microbenchmarks to evaluate specific aspects of CHARON's design and compare it with others. In this section we present results considering a novel benchmark, called FS-BioBench, that we developed to emulate the I/O of common bioinformatics workloads (see Chapter 5).

Bioinformatics I/O benchmark. We analyzed some common workflows executed by bioinformaticians when using data from repositories and developed a set of benchmarks emulating the workload of these tasks on a file system. Table 5.1 summarizes the characteristics of the benchmark tasks. More details can be found in Chapter 5 that describes the design and implementation of FS-BioBench.

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 `W1` 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.

Table 2.4: Characteristics of FS-BioBench workflows. All reads and writes are sequential. Output files are divided in two groups: intermediate and final results. The details about the benchmark can be found in Chapter 5.

Notice the output files of FS-BioBench workflows are divided in two groups: intermediate and final results; and only the latter are written in the evaluated file system. The intermediate results are stored in the same (local) disk that stores the benchmark code. All reads and writes in the benchmark are sequential.

The results. Figure 2.8 presents the execution time of each of the mentioned FS-BioBench workflows for ext4 on SSD (Local-SSD) and disk (Local-HDD) and CHARON using a repository in different locations: SSD in the same machine (C-SSD), hard disk in a server in the same LAN (C-Network), Amazon S3 (C-AS3), and cloud-of-clouds (C-CoC). We executed every workflow six times on each scenario and report average values. The standard deviations were bellow 2%. CHARON's cache was cleaned after each execution.[1]

The Local-SSD and Local-HDD cases serve as basis of comparison in this experiment and, as expected, are always faster in running any benchmark workflow. The time needed to finish workflow `W1` is similar in all CHARON data locations. The same happens in workflow `W2` because write operations immediately return after CHARON updates the file in the local disk,

[1] With cached files, all results will be similar to C-SSD.

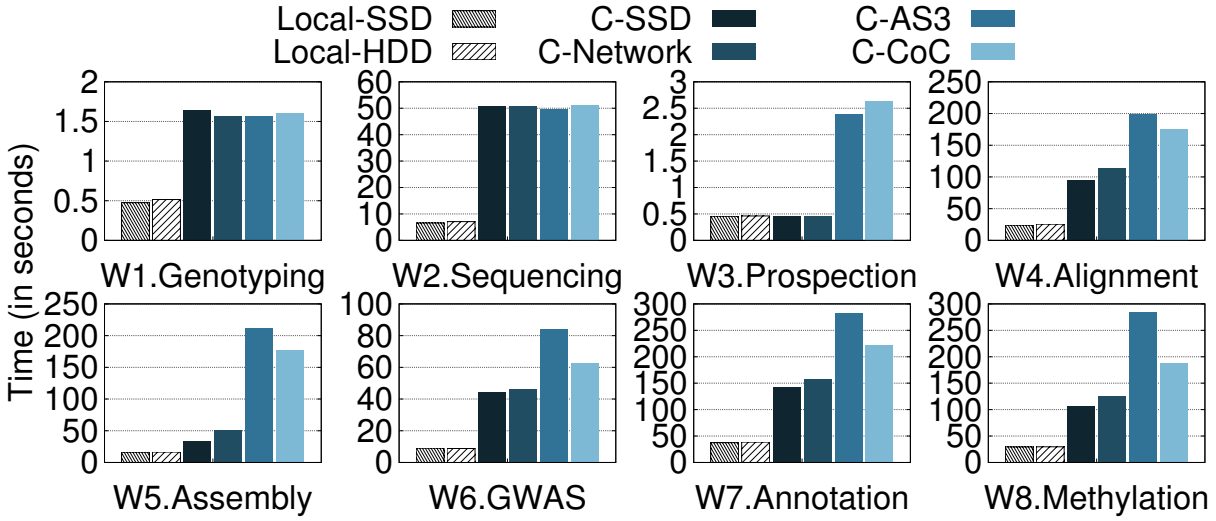


Figure 2.8: FS-BioBench execution for different configurations of CHARON.

since the data is sent in the background to the repository location. Workflow `W3` requires almost $5\times$ more time to finish in C-AS3 and C-CoC than in C-SSD and C-Network due to the latency of fetching the two small files from the remote cloud services. Apart from this, all the other differences between CHARON and ext4 are due to FUSE and the Java wrapper.

Workflows `W4`, `W5`, `W7`, and `W8` are the ones requiring more time to run since they imply the read of a 1GB FASTQ file from the repository location. Workflow `W6` reads two genotyping files with only 24MB each, and thus requires less time to run. Nonetheless, as expected, workflows `W4`-`W8` rank the different data locations in the same order, where the C-SSD is the fastest, followed by the C-Network, the C-CoC, and the C-AS3. Interestingly, running the benchmark in a cloud-of-clouds-hosted repository brings improvements of 13% (`W4`) to 35% (`W8`) in the running time of the benchmark due to the capability of fetching blocks from the two faster clouds at the moment [47].

It is important to remark that, even considering that the latency for fetching input files dominate most of these benchmarks, their workload is sequential, and thus the processing can start as soon as the first block is available.

2.7 Related Work

Distributed file systems. CHARON borrows many ideas from classical file systems, such as the separation of data and metadata from NASD [65], volume leases from AFS [71], and background updates from several peer-to-peer file systems [37, 77, 101]. In particular, Farsite has some similarities with our system, but is crucially different in its use of BFT replica groups for assigning leases and maintaining metadata consistently [37]. Another related system is the xFS [41], a serverless network file system in which all data and metadata storage is done at the client side, through the execution of coordination protocols.

A fundamental difference between these systems and CHARON is that, in the latter clients do not communicate to each other for coordination, and everything is done with the help of widely available untrusted cloud services, which require no server deployment by clients. Furthermore,

these systems does not explore the "infinite" scalability and competitive prices of cloud storage, using instead the storage present on clients and servers. Finally, our design focus on avoiding write/write conflicts due to the expected low level of contention in bioinformatics datasets and to avoid creating conflicts that might be difficult to solve for our target audience (non-system specialists). This decision differentiates our work from recent systems such as OriFS [87], which implements a data model similar to a shared repository with version control.

Data-centric coordination. A centerpiece of CHARON's design is its use of Byzantine-resilient data-centric algorithms for implementing storage and coordination. Besides DepSky (discussed bellow), there are some works that propose the use of this kind of algorithms for implementing dependable systems [35, 57, 84]. At high level, our composite leasing is very similar to the at-most-one mutual exclusion of Phalanx (which is very similar to classical quorum-based mutual exclusion algorithms from 80s). However, their algorithm assumes normal servers, that can run arbitrary code [84], while ours is built on top of non-replicated algorithms especially designed for the cloud services currently available.

To the best of our knowledge, the first proposal for a fault-tolerant data-centric lease algorithm appeared in [57]. This work has two fundamental differences when compared with our Byzantine-resilient lease. First and foremost, it does not provide an always-safe lease, in the sense that their algorithm admits the existence of more than one process with valid leases. The only guarantee is that eventually a single process will have the lease. Second, it only tolerate crashes. These two limitations makes the solution inappropriate for CHARON.

Byzantine disk Paxos [35] is a Byzantine consensus protocol based only on untrusted shared disks (equivalent to storage clouds). This algorithm could be used for implementing mutual exclusion satisfying deadlock-freedom (a stronger liveness guarantee than obstruction-freedom). However, we avoid this solution due to the large number of cloud accesses this protocol might require (at least five per round, and multiple rounds might be needed in case of contention), relying on our composite protocol, which requires at most three sequential cloud access to acquire a lease.

Cloud-backed storage solutions. Table 2.5 summarizes the characteristics of several cloud-backed storage systems with CHARON. The first row presents the models employed on each solution, ranging from archival, key-value stores to full-fledged file systems.

Property	RACS	DepSky	SPANStore	S3QL	Dropbox	SCFS	**CHARON**
Model	Archival	Objects	Key-value	File system	Archival	File system	File system
Private repository	✖	✔	✖	✖	✖	✔	✔
Single cloud	✖	✖	✖	✔	✔	✔	✔
Cloud of Clouds	✔	✔	✔	✖	✖	✔	✔
Serverless	✖	✔	✖	✔	✔	✖	✔
Data sharing	✔	✔	✔	✖	✔	✔	✔
Big data	✖	✖	✖	✔	✔	✖	✔

Table 2.5: Comparison of cloud-backed storage systems.

The next three rows relate to where each system stores its data, namely: private repositories, single cloud, and cloud-of-clouds. Analyzing this, we can see that some solutions use a single cloud service to store the data, e.g., Dropbox [63] stores all data on its facilities and in Amazon

S3. In turn, several solutions avoid vendor lock-in by deploying their systems in a cloud-of-clouds environment, e.g., RACS [36], DepSky [47], SPANStore [112] and SCFS [48]. CHARON is the only system among the ones in the table that supports the storage of data in a private repository, single cloud and cloud-of-clouds.

The next row points to another important aspect to be considered, which is the need for users deploying and maintaining a dedicated service running in public clouds for coordination. RACS uses proxies for encoding and decoding data and Zookeeper for coordinating who can write a given data item [36]. SPANStore uses local and remote VMs, respectively, to lookup metadata and propagate writing operations across the entire system, as well as a central PlacementManager for coordinating locks for objects [112]. SCFS deploys its coordination service in a set of public cloud computing providers like Amazon EC2 and RackSpace Cloud Servers [48]. Additionally, SCFS has an alternative operating mode which is serverless, however it does not support concurrent writes or sharing of information. Dropbox does not require users to deploy a remote service because it already has its own service running for all clients. DepSky is also serverless, however it allows concurrent writes by employing a lease algorithm that uses the public storage clouds [47]. We compared the lease/lock BFT protocol of DepSky with our algorithm in Section 2.6.1 and showed our algorithms is $4\times$ faster than DepSky. Furthermore, contrary to DepSky lease, our composite lease does no require synchronous clocks.

The sixth aspect considered in the table is the support for controlled data sharing, which allows several users to access the same data. RACS implements a one-writer, many-readers approach [36]. DepSky allows many-writers by implementing a lease algorithm which stores the lock information together with the files [47]. SPANStore implements a two-phase locking for write operations [112]. S3QL does not support data sharing [27]. Dropbox provides data sharing through its access control by registering more than one user id to a file or a folder [63]. Additionally, Dropbox employs an optimistic conflict solver, which creates a new file for each conflicted write. SCFS allows controlled data sharing, both for reading and writing, through the use of a coordination service deployed in a public cloud [48]. CHARON allows data sharing by implementing a composite lease, which is stored as metadata together with data.

The last row shows which systems provide specific mechanism for handling large files (e.g., 1GB or larger). S3QL, Dropbox and CHARON are the only solutions that splits files into data blocks to reduce the latency of reads and writes. This optimization allows researchers to analyze data without the need of bringing the entire file before start the computation. All other solutions implement their operations considering the entire file as the data to be read and written by the system.

Integrating biobanks and researchers. Currently, most of the work for integrating biobanks is focused on the definition of a common data format for sharing studies, called MIABIS [92]. Having the same data format for the studies allow the biobanks to store studies metadata in a centralized database, however, it does not provide any support for automated data sharing. Currently, a researcher have to propose a formal project and describe in detail why he/she wants the sample dataset. If approved (a process that can take weeks), he/she is authorized to collect the samples of interest (either physically in an HDD or through a file download). With CHARON, biobanks may work in a federated way, in which once a researcher is admitted to the federation, he/she can access any data set from any biobank, which must be logged for auditing purposes.

There are several research infrastructures aiming for curating bioinformatics datasets. For example, UK Biobank [39] was created to centralize the storage of biological samples from several British biobanks. BBMRI [96] integrates several European biobanks [113] through a

distributed research infrastructure. CGHub [111] is an American research center that stores cancer-related data that can be used by previously-approved users. In all these systems, data is accessed by downloading from web repositories. CHARON works in the file level, which means that it can be used to integrate any type of data, without worrying about the specificities from each bioinformatics application.

2.8 Conclusions

We presented CHARON, a cloud-backed file system for biobank data sharing and storage. The design of CHARON relies on two important principles: files metadata and data are stored in multiple cloud providers and the system is completely serverless. This latter principle lead us to design of a novel Byzantine-resilient leasing protocol for avoiding write/write conflicts. Our results show that such design is feasible and can be employed to interconnect real-world biobanks and other institutions that need to store, archive and share critical datasets in a controlled way.

Chapter 3

Sharing Files Using Cloud Storage Services

Chapter Authors:
Tiago Oliveira, Ricardo Mendes and Alysson Bessani (FFCUL). [1]

3.1 Introduction

With more people accessing their files online, an important part of file sharing today is done by taking advantage of cloud storage. This can be done through personal file synchronization services like Dropbox [63], Google Drive [12], Microsoft OneDrive [21], Box [7] or Ubuntu One [29], which store users' data in the cloud. These services have been extremely successful, as attested by the success of DropBox, which has announced last April that it reached 275 million users [8].

These systems perform file sharing through dedicated application servers which are responsible for controlling access to the files as well as user groups management, data deduplication, etc. It means that the security of the file sharing requires trusting not only the storage service (in the case of Dropbox, it is implemented on Amazon S3 [63]), but also these application servers.

An alternative for using these services is to mount the cloud storage service (e.g., Amazon S3) in a user-level file system and access it directly. S3QL [27], BlueSky [106], SCFS [48] and CHARON (described in Chapter 2) are examples of this kind of systems.

BlueSky uses a proxy that acts as a network file server, which is accessed by the clients in order to store their data. This proxy is responsible for sending the users' data to the storage clouds. Nonetheless, as in synchronization services, clients need to trust this component and the cloud storage provider.

On the other hand, S3QL, SCFS and CHARON allow clients to share data without a proxy. In S3QL there is no control in the file access: the clients mount the storage service objects as files that can be read and concurrent access need to be avoided by clients. SCFS and CHARON, by the other hand, offer controlled file sharing where concurrent updates and file version conflicts are avoided through the use of locks. Moreover, in SCFS and CHARON clients are able to take advantage of DepSky [47] to store data in a multiple cloud providers, i.e., a *cloud-of-clouds*. DepSky, and consequently SCFS and Charon, ensure the privacy, integrity and availability of the data stored in the clouds as long as less then a third of the cloud providers are faulty.

[1] Content of this chapter was previously published in Oliveira *et al.* 2014 [93]

SCFS and CHARON use a model in which each user pays for the files it creates, the *pay-per-ownership* model. A simpler model where all the clients use the same cloud account could be used, and automatically share all the data stored by the system. This alternative raises some problems. First, all users could access all data stored in the clouds. In this way, each client must trust all the system users since they can access, delete or corrupt all stored data. Second, just one organization will be charged for all the data stored in the system.

In this chapter we present a survey of the access-control techniques provided by these cloud storage services. We also show how to implement secure data sharing using these services, allowing the implementation of equivalent data sharing features in different clouds. These protocols were used for supporting the security model of CHARON (see Section 2.5.5).

In summary, we contribute with (1) a study of several cloud storage services' access control models (i.e., Amazon S3 [3], Google Storage [13], Window Azure Storage [30], RackSpace Cloud Files [23], HP Public Cloud [15], and Luna Cloud [17]), i.e, a study of the techniques used by these services to apply the permissions they provided and (2) a set of protocols that allow the sharing of files between clients according with predefined access control patterns for each of the studied storage clouds.

3.2 Access Control on Storage Clouds

To allow users to share their data, all cloud storage services provide some mechanisms that enable data owners (users) to grant access over their resources to other principals.

In all these storage services, the resources can be either *buckets* or *objects*. A bucket, or *container*, represents a root directory where objects must be stored. There could be several buckets associated with a single cloud storage account.[2] However, in most of the services, buckets must have unique names. Objects are stored in a bucket and can be either files or directories.

On the other hand, the cloud storage services differ in the techniques they provide to allow users to grant access over their resources, and also in the permission types that users are able to specify. These techniques will be called *access-granting* and the permission types that can be specified with them will be named *permissions*.

Access-granting techniques. These are the techniques provided by cloud storage services to allow users to give others access to their resources. The users are able to specify a set of permissions in each technique. To apply these permissions over the resources, different storage clouds could offer different techniques. We cover three of them: *Per Group Predefined Permissions*, *Temporary Constraints* and *Access Control Lists* (ACLs).

In the first one the users are able to make their stored data accessible to some predefined group. The second technique allows users to give other users a ticket that grant access for a resource by a predefined period of time. The last one, ACLs, permits to associate with each resource a list of grantees that are able to access it.

Permissions. When a user wants to share some resource with another user, i.e., with a different account/user, it needs to specify what are the capabilities of this other user with respect

[2]Some storage clouds has a limited number of buckets. For instance, an Amazon S3 account can have at most 100 buckets.

to the shared resource(s). Permissions are specified in the access-granting techniques. Each permission has a semantic that specifies the capabilities of the grantees of some resource.

However, different storage clouds provide a different set of specifiable permissions. For example, Amazon S3's users cannot grant WRITE permission to specific objects. By the other hand, RackSpace Cloud Files users can do that [23] (see Section 3.3).

Moreover, equal permissions could have different semantics. As an example, in Amazon S3 [3] when a READ permission over a bucket is given, grantees can list objects inside it. On the other hand, the same permission in Windows Azure [30] does not allow grantees to list the objects in a bucket, instead it grants the permission to read all the objects it contains.

3.3 Permissions

To allow users to define the grantees' capabilities over a shared resource, all clouds provide a set of permissions with documented semantics. As explained before, the same permission could have different semantics in different clouds. Table 3.1 shows the available permissions for buckets and objects in several cloud storage providers.

As can be seen, Amazon S3 [3] and LunaCloud [17] provide the largest set of permissions among all the services studied. They permit users to give READ, WRITE, READ_ACP, WRITE_ACP and FULL_CONTROL permissions [4, 18] over both buckets and objects. Google Storage [13] have almost the same set of permissions. The difference is that Google Storage does not allow users to apply READ_ACP and WRITE_ACP permissions separately [11]. Instead, it put together these two permissions into the FULL_CONTROL one. This means that if a user wants to give other users the capability of read some resource's ACL, it is forced to also grant the capability to write or update that ACL.

Interestingly, in most clouds the READ permission over a bucket does not allow a grantee to read an object inside it. Instead, it only allows grantees to list the objects inside the bucket. To grant read access, a READ permission need to be applied on the desired objects. On the other hand, the WRITE permission on the bucket allows a grantee to write, overwrite or delete any object inside that bucket. In this case, the same permission is not applicable to objects. Given that, it is impossible to grant WRITE access to a subset of the objects inside a bucket. This means that there is no way to grant write access over objects individually.

Another important thing to highlight is that those clouds allow users to give others the right to read or write a resource's ACL through the READ_ACP, WRITE_ACP and FULL_CONTROL permissions. It is also important to notice that when a grantee have the permission to update an ACL, he/she is able to grant access over it to other users without being the resource owner.

HP Public Cloud [15] and RackSpace Cloud Files [23] available permissions are more simple [16, 22]. These two services only provide two different permissions, either for buckets and objects: READ and WRITE. The only difference between these two storage clouds is that the READ permission on the bucket for Hp Public cloud allow grantees to list and read its objects (contrary to Amazon S3, LunaCloud and Google Storage), while for RackSpace only allow grantees to read the objects inside it. Also different of Amazon S3, LunaCloud and Google Storage, in these two clouds is impossible give other users the right to read/update the bucket permissions.

The Windows Azure Storage's [30] set of permissions [31] differs from all other studied cloud storage services. Basically, the WRITE and DELETE permissions are separated, as well as the READ and the LIST. In the other clouds these permissions are grouped in one permission, i.e., when the WRITE access is granted, grantees have also the capability to delete resources.

	Available Permissions	
	On bucket:	**On object:**
Amazon S3	•READ: List the objects in the bucket. •WRITE: Create, overwrite, and delete any object in the bucket. •READ_ACP: Read the bucket ACL. •WRITE_ACP: Write the ACL for the applicable bucket. •FULL_CONTROL: READ, WRITE, READ_ACP, and WRITE_ACP permissions on the bucket.	•READ: Read the object data and its metadata. •WRITE: Not applicable. •READ_ACP: Read the object ACL. •WRITE_ACP: Write the ACL for the applicable object. •FULL_CONTROL: READ, READ_ACP, and WRITE_ACP permissions on the object.
Google Storage	•READ: List a bucket's contents. •WRITE: List, create, overwrite, and delete objects in a bucket. •FULL_CONTROL: READ and WRITE permissions on the bucket. It also lets a user READ and WRITE bucket ACLs and other metadata.	•READ: Download an object. •WRITE: Not applied. •FULL_CONTROLL: READ access. It also lets a user READ and WRITE object ACLs and other metadata.
HP Public Cloud	•READ: Read and list any object in the bucket. •WRITE: Create, overwrite and delete any object in the bucket.	•READ: Read the specified object. •WRITE: Write in the object.
RackSpace	•READ: Read any object in the bucket. •WRITE: Create, overwrite and delete any object in the bucket.	•READ: Read the specified object. •WRITE: Write in the object.
Windows Azure	•READ: Read any object in the bucket. •WRITE: Write and overwrite any object in the bucket. •DELETE: Delete any object in the bucket. •LIST: List the objects in the bucket.	•READ: Read the specified object. •WRITE: Write the specified object. •DELETE: Delete the specified object.
LunaCloud	•READ: List the objects in the bucket. •WRITE: Create, overwrite, and delete any object in the bucket. •READ_ACP: Read the bucket ACL. •WRITE_ACP: Write the ACL for the applicable bucket. •FULL_CONTROL: READ, WRITE, READ_ACP, and WRITE_ACP permissions on the bucket.	•READ: Read the object data and its metadata. •WRITE: Not applicable. •READ_ACP: Read the object ACL. •WRITE_ACP: Write the ACL for the applicable object. •FULL_CONTROL: READ, READ_ACP, and WRITE_ACP permissions on the object.

Table 3.1: Available permissions.

Similarly with HP Public Cloud and RackSpace, a grantee cannot update/read the bucket permissions.

3.4 Access-granting Techniques

3.4.1 Per Group Predefined Permissions

Using Per Group Predefined Permissions, the users are able to apply permissions on buckets or objects granting access for two kinds of groups: *All Users* and *Authenticated Users*. The All Users group refers to anyone in the internet. In turn, the Authenticated Users group represents all users that have an account in the cloud provider. However, to give permissions to these groups, the owners must use some predefined permissions that the storage clouds provide. For instance, Amazon S3 and LunaCloud call this technique *Canned ACLs*, while Google Storage name it *Predefined ACLs*.

The first column of Table 3.2 shows the available predefined permissions for each group, as well as the type of access that each one grants. As we can see, Amazon S3 and Google Storage provide the same Per Group Predefined Permissions [4, 11].[3] These default permissions allow users to make their resources public for the All Users group, for both READ and WRITE. For the Authenticated group, the storage clouds only allow the users to grant READ permission over buckets or objects. Notice that the *bucket-owner-read* and *bucket-owner-full-control* predefined permissions over objects for Amazon S3 and Google Storage, only grant access permissions to

[3]There are some other predefined permissions for this two storage clouds that are not shown in the figure.

resources owners (not for the Authenticated Group). These predefined permissions are provided because the bucket owner could not be the object owner. In these two clouds, each user is the owner of the objects he uploads, even if the uploads are made to a bucket owned by other user. Thus, they are useful to give access rights to the bucket owner when a user uploads an object to a bucket that is not owned by him. The LunaCloud's predefined permissions are similar to the Amazon S3 and Google Storage (see Table 3.2), with the exception that they do not have the *bucket-owner-read* and *bucket-owner-full-control* permissions over the objects.

Windows Azure differs from the previous clouds in two ways. First, there is no predefined permissions to the Authenticated Users group. Second, it provides no way to give WRITE permissions over buckets or objects, allowing only users to grant READ access to the All Users group [32].

Although not shown in the table, all the clouds that provide predefined permissions also provide a special permission that gives FULL_CONTROL to the bucket/object owner, with no one else getting any access to it. In fact, this is the default predefined permission for a resource on its creation.

HP Public Cloud and RackSpace Cloud Files do not provide Per Group Predefined Permissions. However they allow users to make their buckets public through different techniques.

3.4.2 Temporary Constraints

Temporary Constraints is another way to give access permissions to other users. However, using this technique, the access will be temporary. The second column of Table 3.2 shows the studied clouds that implement this access-granting technique. As can be seen, only three of the studied clouds have this feature. RackSpace and HP Public Cloud provide *Temporary URLs* [25, 16], while Windows Azure provide *Shared Access Signatures* [34]. Temporary URLs are used to support the sharing of objects (and only objects), whereas Shared Access Signature allows the sharing of both buckets and objects. These temporary constraints work as a capability given by resource owners to other users in order for them to access to the specified resource. In this case the ticket that proves the right to access the object is the URL. This URL contains information about the period of time that the access will be valid, the path to the resource over which the access is being granted, the permissions granted, and a signature. This signature, not to be confused with a digital signature [95], is different from cloud to cloud:

- RackSpace Cloud Files: SHA-1 HMAC computed over the URL information and a key.
- HP Public Cloud: SHA-1 HMAC computed over the URL information and a key.
- Windows Azure Storage: SHA-256 HMAC computed over the URL information and a key.

In the first case, the key is a sequence of letters chosen by the user, while in the others, it is the secret key used to access the account. This signature ensures (with high probability) that the URL cannot be guessed or changed by an attacker even if he knows the other fields of the URL.

3.4.3 Access Control Lists - ACLs

As shown in Table 3.2, Amazon S3 [4], Google Storage [11], HP Public Cloud [16] and RackSpace Cloud Files [22] are the clouds that allow users to specify access rights to other

	Per Group Predefined Permissions		Temporary Constraints	ACLs
	All Users	**Authenticated Users**		
Amazon S3	•public-read (bucket and object): Owner gets FULL_CONTROL. The All Users group gets READ access. •public-read-write (bucket and object): Owner gets FULL_CONTROL. The All Users group gets READ and WRITE access.	•authenticated-read (bucket and object): Owner gets FULL_CONTROL. The Authenticated Users group gets READ access. •bucket-owner-read (object): Object owner gets FULL_CONTROL. Bucket owner gets READ access. •buclet-owner-full-control (object): Both the object owner and the bucket owner get FULL_CONTROL over the object.	✖	✔
Google Storage	•public-read (bucket and object): Owner gets FULL_CONTROL. The All Users group gets READ access. •public-read-write (bucket and object): Owner gets FULL_CONTROL. The All Users group gets READ and WRITE access.	•authenticated-read (bucket and object): Owner gets FULL_CONTROL. The Authenticated Users group gets READ access. •bucket-owner-read (object): Object owner gets FULL_CONTROL. Bucket owner gets READ access. •buclet-owner-full-control (object): Both the object owner and the bucket owner get FULL_CONTROL over the object.	✖	✔
HP Public Cloud	✖	✖	TempURL	✔
RackSpace	✖	✖	TempURL	✔
Windows Azure	•full-public-access: All Users group gets READ and LIST access. •public-read-access-for-blobs-only: All Users gets READ access .	✖	Shared Access Signature	✖
LunaCloud	•public-read (bucket and object): Owner gets FULL_CONTROL. The All Users group gets READ access. •public-read-write (bucket and object): Owner gets FULL_CONTROL. The All Users group gets READ and WRITE access.	•authenticated-read (bucket and object): Owner gets FULL_CONTROL. The Authenticated Users group gets READ access.	✖	✖

Table 3.2: Access-granting techniques.

users through ACLs. Contrary to the use of Temporary Constraints, by using ACLs, the user does not need to give to grantees a capability (a URL like described in Section 3.4.2). In this case the user who wants to share data needs to create an ACL and include the names or ids (depending on the cloud) of the clients whom he want to give access together with the corresponding permissions and associate it with the shared objects.

However, there are some differences among the storage clouds that provide ACLs. One difference between Amazon S3 and Google Storage, and RackSpace and HP Public Cloud is that in the last two, the users can only manage ACLs for containers. This means that it is impossible for a user to associate an ACL with an object. Another difference is that, while Amazon S3 and Google Storage allow users to grant access to a user from a different account, RackSpace and HP Public Cloud only allow they to set an ACL for sub-users.[4]

All clouds that allow sharing across different accounts through ACLs, do not permit buckets to have the same name, even if they belong to different accounts.

[4]A sub-user is a user within an account owned by other user. Such users can be associated to an account by associating with him a username and a password.

3.5 Setting Per User Permissions

Table 3.3 summarizes which clouds implement mechanisms for securely sharing buckets and objects between different users (which is a requirement for implementing the cloud-of-clouds models of DepSky [47] and SCFS [48]). Among the studied clouds, LunaCloud is the only one that does not provide enough features for this, since it only provides Per Group Predefined Permissions. In the remaining clouds, the per user sharing can be done through ACLs or Temporary Constraints. However, none of these clouds provide mechanisms for securely sharing a bucket in a simple way.

To clarify what we mean by "securely" and "simple", we define a minimum set of rules to share a bucket as:

- ***Rule A***: the permissions on the bucket allow a grantee:

  to list all objects inside it;

  to delete or create any object inside it;

  to read or write any object inside it;

- ***Rule B***: only grantees and the bucket owner can operate on the bucket.

- ***Rule C***: a grantee cannot delegate access rights to other users.

Sharing a container using a cloud storage service that satisfies these rules and support ACL as access-granting technique would be very simple. First, the bucket owner gathers the ids of the accounts he wants to grant access to, and then he creates/associates a bucket with an ACL granting the desired permissions for those accounts.

Unfortunately, as described before, this simple protocol cannot be applied to any cloud we are aware of. However, equivalent functionalities can be implemented in most clouds, albeit using additional steps. In the following subsections we present the steps required for sharing a bucket with specific users in the different clouds in which this is possible.

		Amazon S3	Google Storage	HP Public Cloud	RackSpace	Windows Azure	LunaCloud
Per User Permission	**on bucket:**	ACL	ACL	ACL	ACL	Temp. Constraints	✖
	on object:	ACL	ACL	Temp. Constraints	Temp. Constraints	Temp. Constraints	✖

Table 3.3: Per user permissions.

3.5.1 Sharing with Amazon S3 and Google Storage

Sharing a bucket among specific users in Google Storage and Amazon S3 is quite similar. In the following we describe a protocol (illustrated in Figure 3.1) for sharing a bucket in these storage clouds.

1. The bucket owner needs to gather the ids of the users he want to share with. In the case of Amazon S3, this is the *Canonical User ID* while for Google Storage this is the email associated with the account.

2. The bucket owner must create the bucket and associate with it an ACL with the ids of grantee X and Y granting READ and WRITE permissions. The bucket owner can always get the bucket ACL from the cloud, add more or remove users to it, and update it again.

3. All the ids, including the id of the bucket owner, must be sent to all grantees.

4. When a grantee or the bucket owner uploads an object, it needs to associate an ACL granting READ access to the other grantees (including the bucket owner).

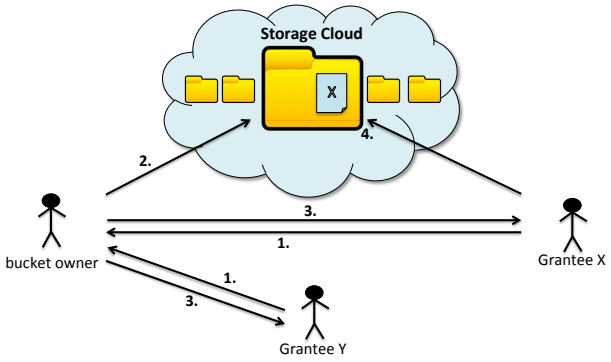


Figure 3.1: Sharing a bucket with Amazon S3 and Google Storage.

The permissions provided by these two clouds are different from the set of permissions we defined in Section 3.5, therefore, this protocol is more costly than the protocol exemplified in that section. For instance, *Rule A* is not respected in the second step: there is no permission on the bucket that allows grantees to read all objects inside it. This leads to the third and fourth step described above: when an object is uploaded to the shared bucket we need to associate an ACL with it to ensure READ access to all grantees.

Assuming that all grantees are trusted, this algorithm fulfill all the requirements present in the set of rules we defined in Section 3.5. However, if any of them deviate from the protocol, some new issues can arise. In these clouds each user is the owner of the objects he uploads, consequently, they can grant READ access over their objects to other users without the knowledge of bucket owner, or even give no access to other grantees. The first case does not respect *Rule C*, while the last case contradicts *Rule A*. Notice that it is impossible to the grantee to give write access to others because these two clouds provide no WRITE permission for objects. If the bucket owner detect these situations, it can always delete the objects the grantee uploaded and revoke his access permissions.

3.5.2 Sharing with HP Public Cloud and RackSpace Cloud Files

HP Public Cloud and RackSpace are the only two clouds, of the five we studied, that allow per user permissions and that provide ACLs and Temporary Constraints to share resources. However, their temporary URLs only allow users to share objects, not buckets (see Section 3.4.2). Figure 3.2 illustrates the steps required for sharing a container using ACLs.

1. The bucket owner needs to get the grantees' names and emails. This is the information needed to add a sub-user. Notice that the grantees do not need to have an HP Public Cloud or RackSpace account.

2. The bucket owner adds the grantees as sub-users of its account. By default, a sub-user cannot access any service until the account owner allows it.

3. After that, the bucket should be created and an ACL with READ and WRITE permissions granting access for the previous added users must be associated with it. For RackSpace in particular, there is no way to update an ACL already associated with a bucket in the cloud, only to replace it. This means that if the bucket owner updates an ACL only granting access to grantee X, when updating the ACL for giving access to grantee Y, the access to grantee X must be granted again.

4. The next step is to provide to grantees the credentials they need to get authenticated as sub-users. The bucket owner can get these credentials after adding the grantees to its account (step 2).

5. From now on, the grantees can authenticate themselves with cloud service using the referred credentials.

6. Thereafter, they can operate in the bucket that was granted access.

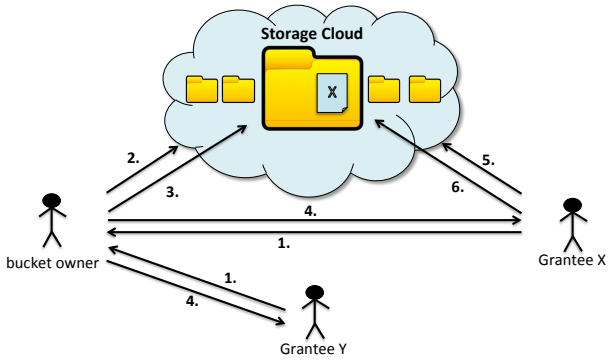


Figure 3.2: Sharing a bucket with HP Public Cloud and RackSpace Cloud Files.

Since RackSpace do not allow grant list access to grantees (see Section 3.3), in this case *Rule A* is not respected. However, in the case of HP Public Cloud all rules that we specify in Section 3.5 are covered. Despite that, in both cases *Rule B* and *Rule C* are respected as long as all grantees are trusted. Otherwise, a non-trusted grantee can provide non-authorized users with the access credentials, making them able to read, write, delete and list on the bucket. This contradicts *Rule C*. However, the bucket owner can revoke grantees permissions just by deleting them from the bucket ACL, or even by removing their sub-users.

Another issue is that these two clouds do not allow sharing across different accounts. This increase the number of necessary steps of the protocol. More specifically, there is the need of steps 2, 4 and 5. It is important that the step 4 is executed over a secure connection to ensure that no one can read the access credentials from the network.

3.5.3 Sharing with Windows Azure

The only way to share a container with other users using Windows Azure is through a Temporary Constraint. Figure 3.3 illustrates the required steps.

1. The bucket owner creates the bucket he wants to share.
2. Generate the Shared Access Signature to this bucket with all the permissions that Windows Azure provide: READ, WRITE, DELETE and LIST.
3. Disseminate the URL among the grantees.
4. Once the grantee have the URL, he can use it to access the bucket for READ, WRITE, DELETE and LIST until its expiration.

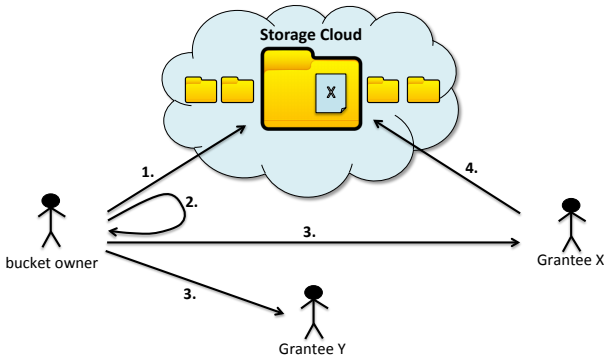


Figure 3.3: Sharing a bucket with Windows Azure.

As in the previous protocols, there is the need to assume that the grantees are trusted. The second step of the protocol allows the creation of a URL with the rules we define in Section 3.5. However, using a Temporary Constraint as an access-granting technique raises some issues. Firstly, step 3 should be done via a secure connection. The second one relates to the fact that a grantee can give to others the URL, thus breaking *rule C*. This means that other users can access the bucket for read, write, list and delete. Finally, there is the inconvenience of repeating the process (with the exception of step 1) every time the URL expires.

3.5.4 Suggestions for Improvements

In each cloud service there are some aspects that can be modified in order to make secure sharing easier. Here they are summarized.

Amazon S3 and Google Storage. As explained above, when a user wants to give the capability of read the content of the files inside a bucket, it needs to give the READ permission to each object inside that bucket. This obviously does not scale in applications with a large number of objects. To solve this problem, these services should provide bucket permissions that give users read access to all objects inside that bucket. In order to prevent grantees to give others access to files they upload, these clouds just need to use a model where the bucket owner is the owner of all objects inside the buckets it pays for, instead of the model where the owner of an object is the user who uploads it.

RackSpace Cloud Files and HP Public Cloud. These clouds require additional steps for defining the credentials for grantees to have access to the shared data. These steps are needed because neither of these services allow cross-account sharing, and could be avoided if this was supported. Specially for RackSpace Cloud Files, a permission to grant list access to grantees should be provided.

Windows Azure Storage. With Shared Access Signatures a malicious grantee is always able to give others the URL allowing anyone to access the shared resources. To solve this issue, Windows Azure just need to provide other access-granting technique, such as ACLs, allowing users to share data between accounts.

3.6 Conclusion

This Chapter presents a study of the access control capabilities of some storage cloud services that permit users to share data using the unmodified clouds directly and assuming a model where each user pays for the storage of the data it stores. The storage clouds studied were Amazon S3, Google Storage, HP Public Cloud, RackSpace Cloud Files, Windows Azure Storage and Luna Cloud.

We described the permissions provided by the services, their semantics, and the different access-granting techniques that are used to apply these permissions to specific users. Additionally, a set of protocols for sharing data securely in several public storage clouds were presented. These protocols were defined by extending an ideal set of properties required for sharing data between different users of a cloud service.

We concluded that none of the studied cloud services offer the tools to implement an optimal solution that respect all these properties, but it is possible to implement sharing in most of them.

Chapter 4

Byzantine-resilient Composite Leasing – Formalization and Correctness Proofs

Chapter Authors:
Ricardo Mendes and Alysson Bessani (FFCUL).

4.1 Introduction

A key innovation of CHARON is a novel fault-tolerant data-centric lease algorithm that exploits different cloud services currently available for implementing multi-cloud leasing. These leases are employed to control concurrent accesses to files and directories, avoiding write/write conflicts. In this way, the performance and reliability of the lease algorithm is imperative to ensure the user experience and the safety of the system are not compromised.

The proposed algorithm uses several non-fault-tolerant base lease objects, which can be implemented in different cloud services, to guarantee the protocol behave correctly in presence of faulty cloud services (including malicious ones). Figure 4.1 illustrates the idea.

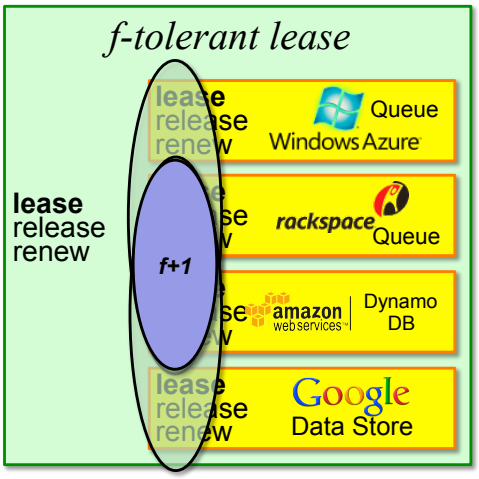


Figure 4.1: BFT-Composite lease strategy.

The proposed algorithm is able to tolerate up to f Byzantine-faulty base objects out-of $n = 3f + 1$ employed. Moreover, it permits users to develop their own *lease* algorithms as

base lease objects and use them in the Byzantine-resilient leasing protocol without change the algorithm itself. This improves the flexibility of the leasing protocol by separating the protocol and the base objects implementations.

Related work. Different from previous works on leases [57, 66], the guarantees of our leasing protocol are more related with the definition of the classical (always safe) mutual exclusion problem [42], in which there can be never more than one process accessing the critical section of its code (i.e., accessing a resource). We extend this definition by introducing the validity of a lease, or *lease term* [66]. As far as we know, there is only one other work in literature that present a Byzantine-fault-tolerant leasing protocol using fail-prone cloud services [47]. However, this algorithm is entirely based on cloud storage services and it needs the clients clocks to be approximately synchronized. Furthermore, as shown in Section 2, the performance of this protocol is much worse than the ones we are formalizing here.

Contributions. This chapter contributes with a formal definition and correctness proofs of the BFT-Leasing protocol and of all developed base lease object implementations (described in high-level in Chapter 2). More specifically, we will describe in detail the lease objects based on *storage services* (Amazon S3 [3], Windows Azure Storage [56], RackSpace Cloud Files [23] and Google Cloud Storage [13]), on *augmented queues* (Windows Azure Queue [20] and RackSpace Queue [19]), and on *Amazon DynamoDB* [2] and *Google Datastore* [10].

Chapter organization. This chapter is organized as follows. Section 4.2 presents our system model and underlying assumptions. Section 4.3 describes our cloud-of-clouds composite lease protocol and its correctness proofs. Finally, Section 4.4 provides the details about the base lease objects employed in CHARON, their formalized algorithms and correctness proofs.

4.2 System Model and Properties

System model. We use the same system model as traditional data-centric Byzantine fault-tolerant algorithms [47]. An unbounded number of clients can access a set of base objects (cloud services) that implement the leasing service. These clients and up to f objects can be subject to arbitrary (or Byzantine) faults. A client can initiate multiple concurrent operation invocations on the same base object, but they are executed in first in, first out (FIFO) order. The base objects implement some form of access control to guarantee that only the clients with the proper permissions can invoke operations. The services used to implement the base objects must provide at least read-after-write consistency.

Timing assumptions. Since the notion of lease implies timing guarantees, an upper bound is assumed for the message transmission time among clients and base objects. This is required only to ensure the liveness of the protocol as safety is always preserved. We assume also zero drift either in clouds' and clients' clocks.

Lease Definition. The lease service for protecting the access to a resource is invoked through the following operations: *lease(T)* to acquire a lease; *renew(T)* to renew a lease; and *release()* to end the lease. These operations have to implement a protocol with the following properties:

- *Mutual Exclusion (safety):* There are never two correct clients with a lease.
- *Obstruction-freedom (liveness):* A correct client that attempts to obtain a lease without contention will succeed in acquiring it.
- *Time-boundedness (liveness):* A correct client that acquires a lease will hold it for at most T time units, unless the lease is renewed.

The above properties do not preclude a Byzantine client from acquiring a lease and keep renewing it constantly, neither constrain a direct access to the resource (e.g., writing a file) without having the corresponding lease. This is not a significant limitation, as a Byzantine client can damage all files for which it has the necessary access permissions. In the same way, a lease acquired by a client that crashes will be available to other clients after at most T time units (Time-boundedness property).

The obstruction-freedom property is the weakest liveness condition for a leasing/mutual exclusion protocol. We chose to satisfy this condition in CHARON because stronger guarantees typically require a greater number of accesses to shared objects and the fact that write-write conflicts are expected to occur rarely in practical deployments.

4.3 Byzantine-resilient Composite Lease

Contrary to previous approaches for data-centric leases [35, 47], which directly rely on quorums of minimal storage services to establish lease acquisition, we propose an approach in which the lease is defined in two steps. First, *base lease objects* are built using a single service from a specific cloud provider, and then a set of $3f+1$ of these lease objects are composed to implement a *f-fault-tolerant lease object*. Besides the flexibility, this method enables the creation of more efficient lease objects that rely on evolved services (e.g., queues, test-and-set-enabled databases, or transactions) made available by the diverse cloud offerings. Furthermore, it allows us to address the heterogeneity of these services, for instance, with regard to offered interface and guarantees.

The key idea of such *composite lease* construction, as specified in Algorithm 2, is to make a client succeed in obtaining the lease if it manages to get leases from a quorum of $n - f$ non-fault-tolerant base objects belonging to distinct cloud providers.

In order to acquire a composite lease, a client calls in parallel the lease operation in each base lease instance (lines 6-7). After that, it waits for $n - f$ success or $f + 1$ failure responses (line 8). In the first case, the lease is acquired and the operation can return. Otherwise, the client needs to release / cancel all the leases — the granted leases and the leases for which no response has arrived (lines 12-13). Then, it backoffs and tries to acquire the lease again after some time (line 14). This procedure is repeated until $n - f$ success responses are obtained or a certain time has elapsed (not showed in the algorithm for legibility reasons). Additionally, the same algorithm is used for renewing a lease (lines 17-19).

To release a lease, a client could simply wait for it to expire. However, to allow other clients to proceed sooner, it invokes in parallel the $release()$ operation in all base lease objects (lines 20-23).

4.3.1 Composite Lease Protocol Correctness

In this section, we present the correctness proofs for the Byzantine-resilient composite lease. In order to prove the mutual exclusion in the access of some resources, there is the need to

ALGORITHM 2: Composite leasing by client c.

```
1  function lease(time) begin
2      result ← 0;
3      repeat
4          L[0 .. n − 1] ⟵ ⊥;
5          startTime ⟵ localTime();
6          parallel for 0 ≤ i ≤ n − 1 do
7              L[i] ⟵ baseLease_i.lease(time);
8          wait until i : (|{L[i] = true}| ≥ n − f) ∨ (|{L[i] = false}| > f);
9          if |{i : L[i] = true}| ≥ n − f then
10             result ← time − (localTime() − startTime);
11         else
12             for i : (L[i] =⊥) ∨ (L[i] = true) do
13                 baseLease_i.release();
14             sleep for some time;
15     until result ≠ 0;
16     return result;
17 function renew(time)
18 begin
19     return lease(time);
20 function release()
21 begin
22     parallel for 0 ≤ i ≤ n − 1 do
23         baseLease_i.release();
```

precisely define what it means for a process to hold a lease for a given resource.

Definition 1. *A correct client c is said to hold the lease at a given time t for $T' > 0$ time units if it obtains T' as response when executing the lease(T) operation in a quorum of base lease objects.*

With this definition, we proceed to prove the properties of Algorithm 2. To prove some of these properties the function C() is used. This function maps the local clock values in the processes to the (global) real-time clock value. Note that the processes do not have access to this function, this is only a theoretical device to reason about the correctness of the protocols.

Theorem 1 (**Mutual Exclusion**). *There are never two correct clients with a lease.*

Proof. Assume this is false: there exists a global real-time instant t in which two clients c_1, c_2 both hold a lease. We will prove that this assumption leads to a contradiction. Let t_1 (resp. t_2) be the local time in which the *lease(T)* returned the value T' to c_1 (resp. c_2), i.e., the moment the client holds the lease. Let also t_{start1} (resp. t_{start2}) be the local time in which the *lease(T)* operation is called by the c_1 (resp. c_2). The moment c_1 assumes the lease has expired is $v_1 = t_{start1} + T'$ (resp. c_2 and $v_2 = t_{start2} + T'$).

Within these definitions, the existence of t requires that either $\mathrm{C}(t_2) \leq \mathrm{C}(v_1)$ or $\mathrm{C}(t_1) \leq \mathrm{C}(v_2)$.

Since the invocation of *lease(T)* precedes its return and it precedes the moment in which the lease expires in all base lease objects , we have:

$$\mathrm{C}(t_{start1}) < \mathrm{C}(t_1) < \mathrm{C}(v_1) \tag{4.1}$$

$$\mathrm{C}(t_{start2}) < \mathrm{C}(t_2) < \mathrm{C}(v_2) \tag{4.2}$$

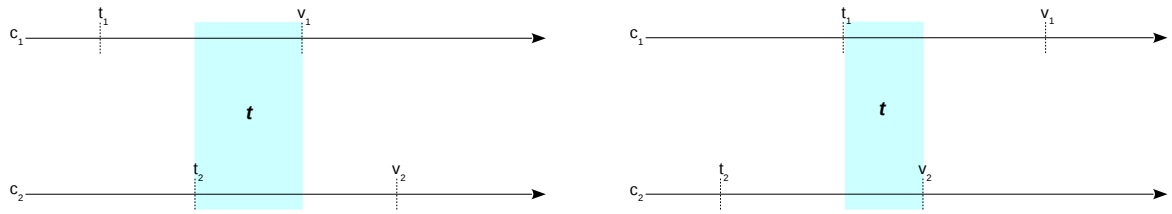


Figure 4.2: Mutual exclusion proof illustration.

Assume the left case presented in Figure 4.2. In this case, since each of the base lease objects guarantees *mutual exclusion* , the client c_2 will only be able to acquire a lease when the lease held by c_1 has already expired in at least $n - f$ base objects, i.e, $\mathrm{C}(t_1) + T' < \mathrm{C}(t_2)$. Given this condition together with Equations 4.1 and 4.2 we have:

$$\mathrm{C}(t_1)+T' < \mathrm{C}(t_2) \implies \mathrm{C}(t_{start1})+T' < \mathrm{C}(t_2) \implies \mathrm{C}(t_{start1}+T') < \mathrm{C}(t_2) \implies \mathrm{C}(v_1) < \mathrm{C}(t_2) \tag{4.3}$$

Equation 4.3 contradicts the left case of Figure 4.2 for the existence of t. The same approach have to be used, assuming that c_2 obtains the lease before c_1, to prove that the other condition is also impossible.

□

Theorem 2 (**Obstruction-freedom**). *A correct client that attempts to obtain a lease without contention will succeed in acquiring it.*

Proof. Client c starts by trying to obtain $n - f$ successful leases from the base objects (lines 5 and 6). It will obtain the lease from all correct base lease objects since, (1) there is no other client holding the same lease, (2) no other client is trying to obtain the lease at the same time, and (3) each base lease object must guarantee the "*Obstruction-freedom*" property. The client c acquires the lease and returns it after obtaining $n - f$ successful responses from base objects. □

Theorem 3 (**Time-boundedness**). *A correct client that acquires a lease will hold it for at most T time units, unless the lease is renewed.*

Proof. Assume that it is false: there is a case in which a client c holds a lease for $T' > T$ time units. We will prove that this assumption results in a contradiction. Let t_1 be the local time *lease(T)* returns the value T' to c_1, i.e., the moment the client holds the lease. Let also t_{start} be the local time in which the *lease(T)* operation is called by c_1.

Given that c_1 holds a lease for $T' > T$ time units and since the invocation of *lease(T)* precedes its return ($t_{start} < t_1$), we have:

$$T' = T-(t_1-t_{start}) > T \implies -(t_1-t_{start}) > 0 \implies t_1-t_{start} < 0 \implies t_1 < t_{start} \tag{4.4}$$

The result of Equation 4.4 contradicts the causal definition of t_{start} and t_1. □

4.4 Base Lease Implementations

Base lease objects can be constructed in various ways. For example by resorting to cloud services that have better performance or that support functions with stronger synchronization power [69] (e.g, a direct lock implementation with a compare and swap). In the next subsections, we describe four different base lease object implementations that were developed using services currently available at popular cloud providers.

Although employing different cloud services, all base lease objects were designed following a group of similar techniques, which allow us to deal with malicious behaviour and manage the duration of a lease. First, the lease operation requires the successful creation of various flavours of *lease entries* in the cloud service. Second, clients have to garbage-collect the outdated (or invalid) lease entries that they create to avoid wasting resources. In most implementations, this cleanup requires at least one cloud access in the lease and renew algorithms. Third, all algorithms sign the lease entries before writing them to the cloud, to ensure that no cloud can create or corrupt leases. Fourth, malicious clients can only attempt to corrupt the leases of resources for which they have the necessary permissions, and therefore they can only hurt themselves (or eventually partners that gave them access to the resource). Lastly, clients do not add a timestamp based on local clock to the lease (to mark its starting period), instead they rely on the cloud service clock to add a timestamp to the created entries. In the same way, clients do not rely in their clocks to check the period of validity of a lease, but instead get from the cloud the current time (which is automatically returned in every operation).

In all algorithms in this chapter, the functions $sign(lease\_id, K_{r_c})$ and $verify(lease\_id, K_{u_c})$ represent obtaining and verifying the signature of the given $lease\_id$, respectively. The function $cloud.getTime()$ obtains the actual cloud time while the function $lastTimeModified(e)$ obtains the cloud time in the instant of the lease entry creation or last renew.

4.4.1 Storage Services

Object storage is one of the most common and popular service made provided by public cloud clouds. We designed an algorithm that implements base lease objects in the same line as the one developed for DepSky [47], but without relying on synchronized clocks among clients and adapted to a single cloud provider (instead of a cloud-of-clouds). The algorithm works in services like Amazon S3 [3], Google Storage [13], Azure Blob Storage [56] and Rackspace Files [23], because they all have a key-value store [46] interface with strong consistency guarantees for functions related to object creation. Algorithm 3 depicts our base lease object implementation using this kind of services.

In our algorithm, to obtain a lease, the client first lists a lease container (line 3) and, if no valid lease entry is found (lines 7-12), it inserts a new one in the container (lines 13-16). To complete the operation, the client lists again the container to be sure that no other lease entries were inserted in meanwhile (together with its own) (line 17). If there is contention and the client finds in any of the two lists another valid lease entry, it removes its own (if already inserted) and returns $false$ (lines 18-20). Otherwise, the lease acquisition succeeds (line 23).

To renew a lease, the client first adds a new lease entry and then removes its old entry (lines 21-22). The release operation corresponds only to the removal of the lease entry.

We depend on cloud providers offering three operations to implement the storage cloud base lease object. The *list()* operation, which returns all the entries within the leases container. The *write(*$name, content$*)* operation which creates a new entry in that container with the name and content given. The *delete(*$name$*)* operation which deletes the entry with the given name.

ALGORITHM 3: STORAGE CLOUD leasing by writer c.

```
1  function storageCloudLease(time)
2  begin
3      L ⟵ cloud.list();
4      cloudTime ⟵ cloud.getTime();
5      lease_id ←"lease-"+c+"-"+time;
6      index ⟵ 0;
7      foreach lease-c'-T'_i ∈ L do
8          if lastTimeModified(lease-c'-T'_i) + T' > cloudTime ∧ verify(lease-c'-T'_i, K_{u_c'}) then
9              if c' = c then
                   // it is a renew
10                 performLease(lease_id,lease-c'-T'_i);
11             else
12                 return false;
13     return performLease(lease_id, ⊥);

14 procedure performLease(lease_id, oldLease)
15 begin
16     cloud.write(lease_id, sign(lease_id, K_{r_c}));
17     L ⟵ cloud.list();
18     if ∃e ∈ L : e ≡lease-c'-T' : c' ≠ c ∧ lastTimeModified(e)+T'> cloudTime ∧ verify(lease-c'-T', K_{u_c'}) then
19         cloud.delete(lease_id));
20         return false;
       // verify if it is a renew
21     if oldLease ≠⊥ ∧oldLease ≠ lease_id then
22         cloud.delete(oldLease);
23     return true;
```

Correctness Proof

In this subsection we will prove the correctness of the cloud storage based lease object implementation presented in Algorithm 3.

Lemma 1. *An entry e created with the operation write(e, s) and not removed, will appear in the result of later list() operations executed on the same cloud.*

Proof. The *write(e, s)* operation is only completed when the entry is created/written in the cloud (line 16 of Algorithm 3). Since we assume that all services used to implement all base lease objects provide at least read-after-write consistency, it means that if a client tries to list the entries in the leases container of some storage cloud after the *write(e, s)* operation, it must return the entry e with the content s written before. □

To prove the properties mentioned before we need to define what it means for a client to hold a lease.

Definition 2. *A correct client c is said to hold the lease at a given time t if an entry $e \equiv$ lease-c-T containing sign(e, K_{r_c}) with lastTimeModified(e)+$T > t$ appears in list() result when this operation is executed in a cloud storage service.*

Now, the safety and liveness properties for Algorithm 3 will be proved.

Theorem 4 (**Mutual Exclusion**). *There are never two correct clients with a lease.*

Proof. Let us assume this is false: there is a time t in which two correct clients c_1 and c_2 hold the lease. We will prove that this assumption leads to a contradiction. If both c_1 and c_2 hold the lease, both *lease-c_1-T_1* and *lease-c_2-T_2* with *lastTimeModified(lease-c_1-T_1)+$T_1 > t$*

and *lastTimeModified(lease-c_2-T_2)+$T_2 > t$* are returned in *list()* from a cloud storage service. Algorithm 3 and Lemma 1 state that it can only happen if both c_1 and c_2 wrote valid lease entries (line 16) and did not remove them (line 19). In order for this to happen, both c_1 and c_2 must see only their lease entries in their second *list()* on the cloud (line 17).

Two situations may arise when c_1 and c_2 acquire write leases: (1) either c_1 (resp. c_2) writes its lease entry before c_2 (resp. c_1) lists the lease entries the second time or (2) c_1 (resp. c_2) writes its lease entry while c_2 (resp. c_1) is executing its second *list()*.

In the first situation, when c_2 (resp. c_1) lists the leases container to discover what were the written leases, it will see both lease entries and thus remove *lease-c_2-T_2* (resp. *lease-c_1-T_1*), releasing the lease (lines 17-20). Situation (2) is more complex because the start and finish of each phase of the algorithm must be analysed. Consider the case in which c_1 finishes writing its lease entry (line 16) after c_2 executes the second list (lines 17). Clearly, in this case c_2 may or may not see *lease-c_1-T_1* in line 18. However, we can say that the second list of c_1 will see *lease-c_2-T_2* since it is executed after c_1 lease entry is written, which happens, only after c_2 starts its second list, and consequently after its lease entry is written. It means that the condition of line 18 will be true for c_1, and it will remove *lease-c_1-T_1*. The symmetric case (c_2 finishes writing its lease file after c_1 executes the second list) also holds.

In both situations we have a contradiction, i.e., there is no execution and time in which two correct clients hold the lease. □

Theorem 5 (**Obstruction-freedom**). *A correct client that attempts to obtain a lease without contention will succeed in acquiring it.*

Proof. When there is no other valid lease entry in the cloud (i.e., the first list in line 3 does not return any lease entry), c will execute the procedure *performLease($lease\_id$, $\perp$)* in line 13 and will write *lease-c-T* on the cloud storage service. This lease entry will be the only valid entry read on the second list (the condition of line 18 will not hold) since (1) no other valid lease entry is available on the clouds since no other client is trying to acquire the lease and (2) Lemma 1 states that if the lease entry was written, it will be available to read. After this, c acquires the lease by returning $true$ (line 23). □

Theorem 6 (**Time-boundedness**). *A correct client that acquires a lease will hold it for at most T time units, unless the lease is renewed.*

Proof. Let us assume this is false: a client that acquires a lease hold it for $T' > T$ time units without renewing it. We will prove that this assumption leads to a contradiction. Let $t_1 =$ *lastTimeModified(e)* be the moment in which the client acquires the lease (i.e. starts holding it). This is the exact moment moment cloud executes the *write(e,s)* operation (line 16 of Algorithm 3), where e is the lease entry with the content s. If the client holds the lease for $T' > T$ time units then exists an instant v_1 such that $v_1 - t_1 > T$.

Algorithm 3 (line 8) and Definition 2 state that a client holds a lease at instant t_1 if an entry e is returned in a *list()* operation and *lastTimeModified(o)+$T > t_1$*. According with this, we have:

$$lastTimeModified(e) + T > v_1 \implies t_1 + T > v_1 \implies T > v_1 - t_1 \tag{4.5}$$

We obtained a contradiction between Equation 4.5 and the causal definition of v_1. □

4.4.2 Augmented Queues

Several clouds provide queue services for the coordination of processes. Some of these services offer strongly-consistent enqueue/dequeue and list operations in its interface, thus offering an augmented queue shared memory object [69]. In the following, we describe an algorithm for base lease objects build on top of Windows Azure Queue [20] and RackSpace Queue [19]. An important feature provided by both of these services is that they permit clients to specify the validity time for entries in the service. The entry vanish from the queues after that time. Algorithm 4 depicts the implementation of the augmented queue base lease object.

A client acquires a lease when its lease entry is the first valid one in the queue. Note that the head of the queue is the first entry of the result returned by the queue *list()* operation. The first step of the lease operation is to list the queue to test if there are other contenders (line 4). If the queue is empty the client enqueues a signed lease entry (line 10). Then, it lists the queue again to check that its entry is the valid one with the lowest index (lines 11-14). If this is the case, the client obtains the lease (lines 15-17).

If the queue is not empty after the first list, the client verifies if some entry in the queue is valid and not created by itself, in which case the lease acquisition fails (lines 5-8). Otherwise, if the lease entry belongs to the client (e.g, is a renew operation), the client pushes a new lease entry to the queue. Additionally, it removes the entries that were observed in order for its lease to be at the head of the queue. To execute a release, the client only tests if it stills holding the lease and removes the entry if this is the case.

Algorithm 4 depends on the queue service to provide the following operations: a *list()* operation, which returns an ordered view of all entries present in the queue; an *add*(e, T), which inserts the entry e to the tail of the queue and ensures that this entry will be in the queue for at most T time units; and a *delete*(l) that deletes all the entries present in the list l. Besides not being essential, in the queue based lease object algorithm we use the operation *delete*(e) that deletes the entry e from the queue. This operation is provided by both Azure and Rackspace Queues.

ALGORITHM 4: AZURE and RACKSPACE QUEUES leasing by writer c.

```
1  function queueLease(T)
2  begin
3      lease_id ⟵ "lease-"+c+nonce+"-"+sign(lease_id, K_{r_c});
4      L ⟵ queue.list();
5      if ∄e ∈ L : e ≡lease-c'-nonce'-s ∧ c' ≠ c ∧ verify(s, K_{u_{c'}}) then
6          return performLease(L, lease_id, T);
7      return false;

8  procedure performLease(L, lease_id, T)
9  begin
10     queue.add(lease_id, T);
11     L_2 ⟵ queue.list();
12     for 0 ≤ i ≤ size(L_2) do
13         if L_2[i] ≡lease-c'-nonce'-s ∧ verify(s, K_{u_{c'}}) then
14             if L_2[i] = lease_id then
15                 queue.delete(L);
16                 return true;
17             else if c' ≠ c then
18                 queue.delete(lease_id);
19                 return false;
20     return false;
```

Correctness Proof

This subsection presents the correctness proofs for Algorithm 4.

Lemma 2. *An entry e, created with the operation add(e, T) at instant t_{add}, will appear as result of later operations list() at instant t_{list} if the condition $t_{add} < t_{list} < t_{add} + T$ holds.*

Proof. The *add(e, T)* operation is only finished when the entry is created in the queue (line 10 of Algorithm 4). Since the cloud ensures the entry e to be available in the queue for T time units after the *add(e, T)* operation occurs, if a client lists the entries in the queue at that interval, e must be returned in the result. □

To prove the properties mentioned before we are obligate to define what it means for a client to hold a lease.

Definition 3. *A correct client c is said to hold the lease at a given time t if an entry lease-c-T-s is the valid lease entry with the lowest index in the result of list() when this operation is executed in a cloud queue service.*

With this definition, we are now able to prove the safety and liveness of Algorithm 4.

Theorem 7 (Mutual Exclusion). *There are never two correct clients with a lease.*

Proof. We will prove this theorem by contradiction. Assume that there is global real-time t in which two correct clients c_1 and c_2 hold the lease.

If both c_1 and c_2 hold the lease we have that both *lease-c_1-T_1* and *lease-c_1-T_2* are returned in the second *list()* at instant t from a queue service. Algorithm 4 and Lemma 2 state that it can only happen if both c_1 and c_2 wrote valid lease entries (line 10). In this case, both c_1 and c_2 must see their lease entries as the valid ones with the lowest index in their second *list()* on the cloud (lines 11-17).

This is impossible since, independently of the order in which the cloud queue executes the lease entry creation requests from clients c_1 and c_2, only one of the entries will be at the head of the queue, being that entry the one with the lowest index in later *list()* invocations. Given this, there is no execution and time in which two correct clients hold the same lease. □

Theorem 8 (Obstruction-freedom). *A correct client that attempts to obtain a lease without contention will succeed in acquiring it.*

Proof. When there is no valid lease entries in the queue, c will executes *performLease(L, $lease\_id$, $time$)* (line 6) since the condition of line 5 will hold. In this procedure the entry *lease-c-$nonce$-s* will be created on the queue service.

On the second *list()* the only valid lease entry obtained will be the one just written. Due of that, the conditions of lines 14 and 15 will hold since (1) no other valid lease entry is available on the queue because no other client is trying to acquire the lease and (2) Lemma 2 states that if the lease entry was written it will be available to subsequent *list()* operations. At this point, c will acquire the lease after returning $true$ (line 18). □

Theorem 9 (Time-boundedness). *A correct client that acquires a lease will hold it for at most T time units, unless the lease is renewed.*

Proof. Definition 3 defines that a client c holds a lease e if it is the valid lease entry with the lowest index in the result of the operation *list()*.

Since the cloud queue services ensure that an entry e created with the operation *add(e, T)* will be present in the queue for at most T time units, then a lease entry inserted by this operation will also be returned as result of the *list()* operation for at most T time units. □

4.4.3 Amazon DynamoDB

Amazon DynamoDB [2] is a NoSQL database-as-a-service that can be utilized both by internal (i.e., VMs deployed in Amazon EC2) and external clients. The current version of DynamoDB provides strongly-consistent storage and a test-and-set function that enables the implementation of very efficient base lease objects, such as the one described in Algorithm 5.

In this algorithm, the first step to acquire a lease is to search for a lease entry related with a specific resource in the database (line 3). Whenever such lease entry exists, we verify checked the ownership, validity time and integrity. If that lease belongs to another client and is still valid, the operation returns $false$ (lines 5-7). Otherwise, the client creates a new lease entry and uses the database test-and-set function (*testAndSetItem()*) to write the entry (line 12). The function ensures that the new lease entry is set only if in the meanwhile no other client added an equivalent entry, returning $true$ / $false$ in case of success / failure.

To renew (resp. release) the lease, the client uses the the test-and-set function to update the lease entry to extend its duration (resp. to invalidate its state).

The algorithm presented needs the DynamoDB to provide only two operations. First, the *query(field, EQ, value)* operation, which returns the entries in the service that have the value of the given field equals to the given value. Notice that, DynamoDB permits users to use other operators different from "equals" (EQ) but in our algorithm we only use this. Besides DinamoDB allows users to search for entries using all of the entries' fields, our algorithm only uses the "key" field, which is guaranteed to be the entries unique identifier. The second operation is the *testAndSetItem(e', e)*. This operation tests if there is some entry with the same key of e, tests if that entry is equal to e', and if so replaces it with e, returning success. If there is no entry to replace with e, then it is inserted (i.e. the operation succeeds) only if $e' = \perp$.

ALGORITHM 5: DYNAMODB leasing by writer c.

```
1  function dynamoDBLease(time)
2  begin
3      res ⟵ db.query("key", EQ,"lease");
4      cloudTime ⟵ db.getTime();
5      if res ≠⊥ ∧res.cId ≠ c then
6          if res.expirationTime < cloudTime ∧ verify(res, K_{u_res.cId}) then
7              return false;
8      lease.key ⟵"lease";
9      lease.expirationTime ⟵ cloudTime + time;
10     lease.cId ⟵ c;
11     lease.sign ⟵ sign(lease, K_{r_c});
       // only succeeds if value in service is equal to the given (e.g. res)
12     succeed ⟵ db.testAndSetItem(res, lease);
13     return succeed;
```

Correctness Proof

In this subsection we will prove the correctness of the Amazon DynamoDB based lease object implementation presented in Algorithm 5.

Lemma 3. *An entry e with the key "lease" created with a successful operation testAndSetItem(e', e) and not removed will appear as the result of later operations query("key", EQ,"lease").*

Proof. As explained before, the *testAndSetItem(e', e)* operation only is succeeded if the entry is created / updated in DynamoDB service (line 12 of Algorithm 5). If a client tries to read the available lease entry by executing the *query("key", EQ,"lease")* operation after a successful *testAndSetItem(e', e)* operation occurs, clearly the cloud must return e as result. □

To prove the properties mentioned before we have to define carefully what it means for a client to hold a lease.

Definition 4. *A correct client c is said to hold the lease at a given time t if a valid lease entry e with $e.key =$"lease", $e.cId = c$ and $t < e.expirationTime$ is stored in the DynamoDB cloud service.*

With this definition, we are now able to prove the safety and liveness of Algorithm 4.

Theorem 10 (Mutual Exclusion). *There are never two correct clients with a lease.*

Proof. Assume this is false: there is global real-time t in which two correct clients c_1 and c_2 hold the lease.

If both c_1 and c_2 hold the lease, accordingly with Lemma 3 and Definition 4, we have that both must successful execute the *testAndSetItem(e', e)* operation (line 12).

Since this operation is atomic, either if c_1 or c_2 execute this operation first, only one of the two will be succeed in inserting its lease entry e in the cloud service, being it the one that obtains success in the operation. When the other executes that operation, the lease entry present in the cloud will differ from the one it provides as e' and then its lease entry will not be inserted. Consequently, we have a contradiction. □

Theorem 11 (Obstruction-freedom). *A correct client that attempts to obtain a lease without contention will succeed in acquiring it.*

Proof. When there is no contention in acquiring the lease and there is no valid lease entries in DynamoDB, c will obtain $res = \perp$ when executing the *query("key", EQ, "lease")* (line 3). After this, c will try to insert the lease entry e in the service by executing *testAndSetItem($\perp$,e)* (line 12). Since there is no contention in acquiring the lease, this operation will succeed and c will acquire the lease. □

Theorem 12 (Time-boundedness). *A correct client that acquires a lease will hold it for at most T time units, unless the lease is renewed.*

Proof. We will prove this theorem by contradiction: there is an instant t in which a client holds a lease for more than $T' > T$ time units without renew it. Let t_{start} and t_1 be the moment in which the *lease(T)* operation is called and the moment in which the client starts holding the lease, i.e. inserts the lease entry e in Amazon DynamoDB, respectively. Let also $v_1 = t_{start} + T$ be the expiration time of that lease. If the client holds the lease for $T' > T$ time units then we have t such that $t - t_1 > T$.

Since the start of execution of the *lease(T)* operation precedes the acquisition of the lease and its expiration ($t_{start} < t_1 < v_1$) and since Algorithm 5 and Definition 4 state that a client holds a lease in instant t only if $t < v_1$, we have:

$$t < v_1 \implies t < t_{start} + T \implies t < t_1 + T \implies t - t_1 < T \tag{4.6}$$

The result of this equation contradicts the causal definition of t. □

4.4.4 Google Datastore

Google Datastore [10] is also a database service that can be accessed through the internet. Differently from the DynamoDB, it does not have a test-and-set function but supports general atomic transactions [55], which allows the implementation of obstruction-free base lease objects. Algorithm 6 presents our implementation of such object using this service.

The lease operation is within the boundaries of a transaction. The client first performs a lookup for a lease entry in the database service (line 4). If there is a valid lease entry, the transaction is aborted and the operation fails by returning $false$ (lines 8-10). Otherwise, the client inserts its own lease entry and attempts to commit the result (lines 19 and 22). Committed transactions indicate that there were no conflicts and no other lease entries were inserted, and thus the lease operation can return $true$. In the opposite case, the lease acquisition fails with a $false$ result.

The renew operation is implemented approximately in the same way of the lease operation. The only difference is that it updates the lease entry instead of inserting it (line 21). To implement the release, the client opens a transaction and executes a lookup to check if the lease belongs to it. If this is the case, it deletes the lease entry and commits the transaction.

In order to implement the base lease object presented in Algorithm 6, Google Datastore service must provide the operations *beginTransaction()* and *commit(*$trans$*)* which starts and commits a transaction, respectively. The operations *lookUp(*key, $trans$*)*, *insert(*$entry$, $trans$*)* and *update(*$entry$, $trans$*)* are also required. The *lookUp* operation returns the entry present in the cloud service with the provided key (if there is not a correspondent entry it returns $\perp$). The *insert* and *update* operation inserts and updates an entry in the service, respectively. Notice that the *insert* operation only succeeds if when it is committed there is no entry in the service with the same key. In the same way, the commit of an *update* operation only succeeds if there is an entry with a key equal to the the given entry in the service.

ALGORITHM 6: GOOGLE DATASTORE resource leasing by writer c.

```
1  function datastoreLease(time)
2  begin
3      trans ⟵ ds.beginTransaction();
4      res ⟵ ds.lookUp("lease", trans);
5      cloudTime ⟵ ds.getTime();
6      if res =⊥ then
7          return performLease((cloudTime + time), false, trans);
8      if res.cId ≠ c ∧ res.expirationTime < cloudTime ∧ verify(res.sign, K_{u_res.cId}) then
9          ds.commit(trans);
10         return false;
11     return performLease((cloudTime + time), true, trans);

12 procedure performLease(expTime, isRenew, trans)
13 begin
14     lease.key ⟵"lease";
15     lease.cId ⟵ c;
16     lease.expirationTime ⟵ expTime;
17     lease.sign ⟵ sign(lease, K_{r_c});
18     if !isRenew then
           // only succeeds if there is no equal key in the service
19         ds.insert(lease, trans);
20     else
21         ds.update(lease, trans);
22     return ds.commit(trans);
```

Correctness Proof

This subsection presents the correctness proofs of the Algorithm 6.

Lemma 4. *An entry e with the key "lease" created with a operations insert(e, $trans$) and subsequent commit($trans$), if not removed, will appear as result of later operations lookUp("lease",$trans'$) with $trans' = trans \vee trans' \neq trans$.*

Proof. As described before, the *insert(e', $trans$)* inserts the entry e in the Google Datastore if, when the subsequent commit occurs, there is not other entry with the same key of e (i.e. "lease"). If the subsequent *commit($trans$)* operation succeeds it means that the entry was created in the service (line 19 of Algorithm 6). Due of this, if the entry e was not removed it must appear as result of later executions of the *lookUp("lease",$trans'$)* operation either if $trans'$ is the transaction used to create the entry e or not. □

To prove the safety and liveness of the Algorithm 6, we need first to define what it means for a client c to hold a lease.

Definition 5. *A correct client c is said to hold the lease at a given time t if a valid lease entry e with $e.key =$"lease", $e.cId = c$ and $e.expirationTime > t$ is stored in Google Datastore.*

With this definition, we are able to prove the properties for the base lease implementation presented in Algorithm 6.

Theorem 13 (Mutual Exclusion). *There are never two correct clients with a lease.*

Proof. Assume this is false: there is global real-time t in which two correct clients c_1 and c_2 hold the lease.

If both c_1 and c_2 hold the lease, accordingly with Definition 5 and Lemma 4, we have that both must obtain success in the *commit($trans$)* operation after the insertion of the entry e (line 19 and 22).

Since the transaction $trans$ is started before the *lookUp("lease", $trans$)* (at line 3) and is only committed after the insertion of the entry e in the Google Datastore (line 22), this makes these two operation to be executed in an atomic way. Due of this, there are never two clients executing the *lookUp("lease-", $trans$)* and *insert(e,$trans$)* operations concurrently.

In this way, when the *commit($trans$)* operation of c_1 (resp. c_2) is executed, if there is no other client that holds a valid lease, it will succeed and c_1 (resp. c_2) acquires the lease. At its turn, when c_2 (resp. c_1) executes its *commit($trans$)* operation, according with Lemma 4, there will be in the Google Datastore the lease entry previously written by c_1 (resp. c_2). Since the *insert(e, $trans$)* needs the service to not have any entry with the same key of e, the subsequent *commit($trans$)* operation will not succeed, resulting in c_2 (resp. c_1) to failing in acquiring the lease.

We have a contradiction because there is no possible execution and real-time t in which two correct clients hold the lease. □

Theorem 14 (Obstruction-freedom). *A correct client that attempts to obtain a lease without contention will succeed in acquiring it.*

Proof. Since there is no contention in acquiring the lease, the Google Datastore cloud service will not have any valid lease entries and then, when c executes the *lookUp("lease", $trans$)* operation, it will obtain $res = \perp$ (line 4).

After this, it will execute the procedure *performLease(*$expTime$, $isRenew$, $trans$*)* at line 7 of the Algorithm 6 with $isRenew = false$. In this procedure, client c will call the operation *insert(*e, $trans$*)* and will commit the transaction $trans$, being e the lease entry. Since $trans$ started before the *lookUp("lease",* $trans$*)* operation and there is no contention in acquiring the lease, the *commit(*$trans$*)* operation must succeed. According with the Definition 5 it means that c holds the lease. □

Theorem 15 (**Time-boundedness**). *A correct client that acquires a lease will hold it for at most T time units, unless the lease is renewed.*

Proof. This proof is similar to the one from Theorem 12. We will also prove this theorem by contradiction: there is an instant t in which a client holds a lease for more than $T' > T$ time units without renew it. As in that proof, let t_{start} and t_1 be the moment in which the *lease(*T*)* operation is called and the moment in which the lease entry e is inserted in the Google Datastore, respectively. The expiration time of the lease is given by $v_1 = t_{start} + T$. Assuming client holds the lease for $T' > T$ time units then we have t such that $t - t_1 > T$.

Due to the fact that the call of the *lease(*T*)* operation precedes the acquisition of the lease and its expiration time ($t_{start} < t_1 < v_1$) and since Algorithm 6 and Definition 5 state that a client only holds a lease at instant t if $t < v_1$, we can infer:

$$t < v_1 \implies t < t_{start} + T \implies t < t_1 + T \implies t - t_1 < T \tag{4.7}$$

The result of this equation contradicts the causal definition of t. □

4.5 Final Considerations

In this chapter we formalized the algorithms used for leasing in the CHARON file system. We presented the Byzantine-resilient composite leasing protocol and its correctness proofs. This chapter described also the base lease implementations that the composite leasing protocol uses and provides correctness proofs for all of them.

Chapter 5

FS-BioBench: A File System Macrobenchmark from Bioinformatics Workflows

Chapter Authors:
Vinicius Cogo and Alysson Bessani (FFCUL).

5.1 Introduction

Macrobenchmarks measure the overall performance of a system by executing a subset of real programs or representative functions from a specific application area [73]. Bioinformatics workflows usually are data intensive, and the next generation sequencing (NGS) advent will intensify this even more. Surprisingly however, existent bioinformatics macrobenchmarks [38, 43, 79, 91] focus on CPU-bound tasks rather than on I/O-bound ones.

A file system is the underlying structure a computer uses to abstract some hardware device to allow users read and write files instead of data blocks [73]. Benchmarking file systems is complex and certainly contains several pitfalls [102, 103]. Nonetheless, the use of bioinformatics workflows may contribute to this area by inciting empirical reasoning on how to properly manage big data.

In this chapter, we describe the novel file system macrobenchmark, FS-BioBench, employed in the evaluation of CHARON (see Section 2.6.3). FS-BioBench simulates the I/O of representative tasks commonly executed in the NGS life-cycle, where the interactions between bioinformatics researchers and data repositories are the starting point of our study. The simulated tasks were selected based on an analysis of important data-intensive workflows executed in bioinformatics research.

The remaining of this chapter is divided in four sections. Section 5.2 introduces and describes the bioinformatics workflows considered in our benchmark. Section 5.3 presents our tool, and shows the results from useful experiments using it. Finally, Section 5.4 examines the differences between our tool and other bioinformatics macrobenchmarks, and Section 5.5 concludes the chapter with some final considerations.

5.2 Bioinformatics Workflows

Next generation sequencing (NGS) is the advent of high-throughput methods for sequencing entire genomes at an affordable cost. In fact, it is already changing the basic structures of bioinformatics workflows [86]. Earlier algorithms focused on working with small sequences, while more recent proposals are required to work with whole genome sequences, or even large sets of genomes. In this section, we introduce the interactions between bioinformatics researchers and data repositories, and present the workflows considered in our benchmark.

`W1.Genotyping`. Genotyping is the process of obtaining the genomic variations from an individual compared either with another individual or with a reference genome. DNA microarray is one of the most used technologies for genotyping human samples, and is able to genotype millions of variations at once. Each entry corresponding a genotyped variation contains an identifier (e.g., rs9519436), the chromosome and position where it is found (e.g., 13 105301516) and the individual's genotype (e.g., GT). This last field, the genotype, contains two letters corresponding one allele inherited from the father's chromosome and one from the mother's.

Each entry line sizes 27 bytes, and the number of genotyped variations depend on the platform used. For example, genotyping files from 23AndMe [1] contain more than 960k variations and size approximately 24MB. In the `W1.Genotyping` workflow, we sequentially write a text file containing one genotyped variation entry per line up to the number of entries specified by the user.

`W2.Sequencing`. DNA sequencing is the process of obtaining all nucleotides from portions or the entire genome of an individual. However, this process does not directly create a contiguous DNA sequence. Sequencing output usually is divided in several large FASTQ files containing unaligned small DNA reads, which are composed of 50–1000 base pairs each. Each read entry is composed of 4 lines: a comment line about the sequence, the sequence itself, a comment line about the read quality, and the read quality itself, as presented in the following:

```
@SRR067577.2766/1
TATATTGGTCAGGCTGCCTGACCTCAGGCGATCCACCCGCCTCAGCCTCC
+
IFIHHGHIHGIGHHEGGGDEGEEGGDGEGEDEDD@DD7DD@BB@>=B###
```

In the `W2.Sequencing` workflow, we sequentially write a text file in FASTQ format containing read entries composed of those four lines. The workflow receives the DNA read size and the number of sequenced reads from users as arguments. Each entry will size $(2 \times S) + 23$ bytes, where S is the read size. For example, an entry for reads with 50 nucleotides will size 123 bytes, and with 100 nucleotides will size 223 bytes. The final output size will depend also on the number of DNA reads it will contain. For example, the sequencing output of an entire human genome with $30\times$ coverage sizes approximately 180GB.

Workflows `W1.Genotyping` and `W2.Sequencing` are write-only and represent common data storage tasks. Their output files are considered raw data for several other bioinformatics workflows since genotyping and sequencing data are the most stable data one can obtain from biological samples.

W3.Prospection. Finding and selecting the appropriate data samples for a study is an important preparation step for researchers. Large quantities of samples are necessary for several studies. There are estimates that genetic main-effect studies require 2–5k samples, lifestyle main-effect studies require 2–20k samples, and gene-lifestyle interaction studies require 20–50k samples to obtain valid results [54]. Large data storages have an extremely important role in this prospection since they are responsible for providing and facilitating the access to data samples.

MIABIS [92] is a data model proposed as the minimal dataset needed for sharing biobank samples, information and data. The BiobankCloud deliverable D1.2 [80] describes how MIABIS integrates with BiobankCloud project. Our third workflow (`W3.Prospection`) implements the MIABIS data model in XML files and simulates a query for selecting appropriate sample collections, by country and disease, for a study. Each biobank entry in MIABIS XML sizes approximately 900 bytes and a sample collection entry sizes 1400 bytes. The BBMRI-ERIC is a project that has a catalog of European biobanks, which contains registers of 330 biobanks and of more than 700 sample collections. The biobanks XML file in our workflow sizes 256kB and the XML with sample collection entries sizes approximately 850kB. The output file from `W3.Prospection` workflow contains a list of prospected samples that sizes 4kB.

W4.Alignment. Read alignment finds the chromosome and the position in a reference genome where a DNA read is found [78]. Researchers align all DNA reads from a sequenced genome (e.g., from workflow `W2.Sequencing`) before starting several more complex analyses (e.g., `W7.Annotation`). Thus, the input of workflow `W4.Alignment` is composed of sequencing data, which in humans may reach 180GB per file. The aligned reads are stored in a format called Sequence Alignment/Map (SAM), where each entry sizes approximately $(2 \times S) + 134$ bytes, where S is the read size. It is bigger than a FASTQ entry because it contains almost the same information plus the alignment result.

Workflow `W4.Alignment` sequentially reads the FASTQ input file, and writes one SAM entry for each FASTQ entry read (composed of four lines). However, not all DNA reads are successfully aligned because there are either unknown sequences with no good match or ambiguous sequences that could be aligned to more than one chromosome and position. In our benchmark, for each 1GB of FASTQ input read, we write approximately 960MB to the output.

W5.Assembly. Genome assembly obtains the contiguous whole genome sequence from a sequencing data file (e.g., from workflow `W2.Sequencing`) [97]. Each DNA read is first aligned to a genome reference, similarly to worfklow `W4.Alignment`, and finally the resulting fragments are merged in a single contiguous sequence. The entire assembled sequence is written in FASTA format only at the end of workflow `W5.Assembly` because the merging step needs all aligned sequences containing each position to decide on the best match. A human genome in this format sizes approximately 3GB.

Workflows `W4.Alignment` and `W5.Assembly` are considered basic workflows because they are used by several other bioinformatics workflows. Their output files contributes to several complex analyses because knowing where each DNA read is positioned in a genome allows researchers to compare the segments with other studied sequences with well-known functions.

W6.GWAS. Genome Wide Association Studies (GWAS) correlate genomic variations and traits by comparing diagnosed patients (i.e., cases) and healthy people (i.e., controls). Variations that are much more frequent in one group, when compared to the other, become variations of interest. GWAS' main goal is to find good evidences that the presence or absence of a specific set of variations accelerates or inhibits the development of some disease, which will be deeply investigated by other studies with different workflows.

In `W6.GWAS`, we sequentially read two genotyping files (e.g., from `W1.Genotyping`) with approximatelly 24MB each one, and create an intermediate file in the Variant Call Format (VCF). Each line in the VCF file contains the chromosome, the position and the identification of the genomic variation, the reference and the variant alleles, and the genotype of each individual included in this study. It sizes approximately $(4 \times I) + 28$, where I is the number of individuals included in the GWA study. The next step reads the VCF file, and for each genomic variation simulates the calculation of odds ratio of an individual having the mutation in his genome and contracting the studied disease. Additionally, the p-value is calculated for significance of odds ratios. These two variables are simulated and written in a second intermediate file that contains one line per genomic variation and each entry sizes approximately 37 bytes. The third and last step of this workflow writes a bitmap file with the same size as a Manhattan plot (less than 300kB), which correlates the genome positions with the negative logarithmic of the p-value.

W7.Annotation. Attesting the presence or absence of known disease-related genes in a genome is important for suggesting the predisposition to an individual contract a disease [83]. Annotating a genome, with known biological information related to genomic data, accelerates the genomic reports from the personalized medicine context, which are auxiliary methods to medicine rather than diagnoses.

The `W7.Annotation` workflow sequentially reads a sequencing data file (e.g., from the `W2.Sequencing`), and aligns all DNA reads similarly to `W4.Alignment`, and writes a SAM file. The second step of this workflow iterates the SAM file searching for genomic variations and writes a VCF file. The third and final step reads the VCF file, annotates the genomic variations with known biological information, and write them to a new VCF file which is called annotated VCF. In our workflow implementation, this annotated VCF sizes approximately 268MB.

W8.Methylation. The last workflow comes from the epigenetics area, which is an area that studies phenotype changes that are not caused by modifications in the DNA sequence (the genotype). Such changes in gene expression can be caused by external factors, for example, by environmental conditions. A DNA methylation is a natural chemical process that alter the expression of genes, for good (i.e., development) and for bad (i.e., diseases) [70]. A methylation happens when a methyl group is added to a cytosine or adenine nucleotide. Diverse studies are analyzing the effect of methylation in humans, mainly related with development and aging. For example, Heyn *et. al* [70] discovered that there is a global loss of DNA methylation during aging.

The `W8.Methylation` workflow sequentially reads a special version of a sequencing data file, called Whole Genome Bi-sulfite Sequencing (WGBS), aligns all DNA reads similarly to `W4.Alignment`, and writes a SAM file. The second step of this workflow estimates the genotype and methylation status, and creates an intermediate file containing all this information, which sizes approximately 30MB. The third and final step of this workflow writes a list of candidate residues for differential methylation, which sizes 1MB in our case.

Workflows `W6.GWAS`, `W7.Annotation` and `W8.Methylation` are considered complex workflows because they have more than one internal step, and they provide results that are extremely useful for extracting interesting information from genomes. Table 5.1 presents an overview of all those eight workflows with the number, size and format of input and output files, as well as the references for works explaining the approaches in detail. An additional description about some of these workflows can be found in the deliverable D6.2 [72] from BiobankCloud project.

Workflow	Input			Output		
	# Files	Total Size	Format	# Files	Total size	Formats
W1.Genotyping [107]	–	–	–	0 + 1	0 + 24MB	TSV
W2.Sequencing [58, 109, 86]	–	–	–	0 + 1	0 + 1GB	FASTQ
W3.Prospection [92, 110]	2	1MB	XML	0 + 1	0 + 4kB	XML
W4.Alignment [64, 78, 67]	1	1GB	FASTQ	0 + 1	0 + 960MB	SAM
W5.Assembly [64, 114, 97]	1	1GB	FASTQ	0 + 1	0 + 18MB	FASTA
W6.GWAS [88]	2	48MB	TSV	2 + 1	432MB + 200kB	VCF,TSV + BMP
W7.Annotation [83]	1	1GB	FASTQ	2 + 1	1.07GB + 268MB	SAM,VCF + VCF
W8.Methylation [70]	1	1GB	FASTQ	2 + 1	999MB + 3MB	SAM,BED + TXT

Table 5.1: Workflows considered in our benchmark. All reads and writes are sequential. Output files are divided in two groups: intermediate and final results.

5.3 The FS-BioBench

A common framework is necessary to provide all workflows considered in the previous section. The present section contains an overview of FS-BioBench design, as well as its implementation and usage. Finally, we describe some experiments that exemplify the utility of our benchmark for different target-audiences, and alert for some limitations and pitfalls that may affect users.

5.3.1 Overview

We designed the FS-BioBench macrobenchmark mostly with a specific principle in mind: simplicity. The simpler the benchmark is, the easier it is to be installed, configured, modified, run, and interpreted. Concisely, the other design principles employed in our benchmark are:

- *Minimal dependency* on specific software installers, compilers, libraries and packages.
- *Configurable usage* to improve the suitability for different parameters on the same workflow.
- *Interoperability* to compare file systems deployed in several operating systems.
- *Code readability* to make the benchmark easy to be modified or improved.
- *Abstract* file system operations to allow the usage with local and network file systems.

The target-audiences of our benchmark are data repository owners, computational biologists and file system developers. Data repository owners (including biobankers) aim to evaluate their solutions to data storage, prospection and provisioning. Computational biologists aim to

evaluate their solutions to data collection, analysis and sharing. File system developers may aim to evaluate their products in a bioinformatics scenario with a straightforward benchmark.

Our system model considers a single machine with two file systems mounted on it, as presented in the Figure 5.1. The first one is a local file system that stores the operating system (OS) and the benchmark code, and abstracts a storage device in the same machine. The second file system (the Target FS in Figure 5.1) can abstract a local or a remote data storage solution, which will maintain all the input and output data from our benchmark. This separation allows us to minimize the effects from one mount point to another, and makes easier to one analyze different storage devices and file systems with our benchmark.

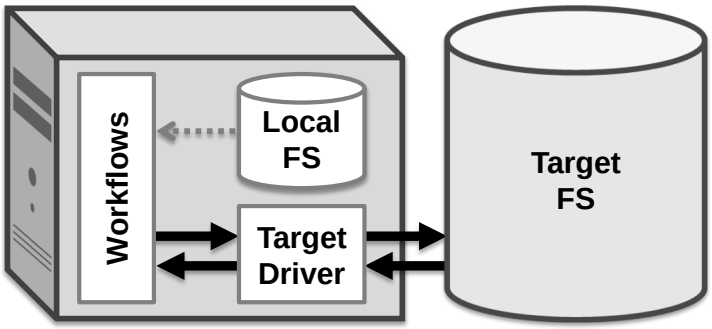


Figure 5.1: Our system model for the FS-BioBench macrobenchmark.

Some other design aspects are noteworthy. First, we focus on I/O tasks rather than on processing ones, which reflects the choice of replacing common CPU-intensive tasks by *dummy* processes that consume data as fast as possible. Second, we read and write synthetic data since our processing tasks ignore the meaning of data, but we use real file formats and sizes. Third, no input data is stored in the local file system (see Figure 5.1) since it would only compromise benchmarking the Target FS. Fourth, no reference genomes or databases are used in our benchmark because there is no real processing and this data would be obtained from the local file system or the Internet in execution time. Fifth, our benchmark consumes only one file per workflow step at time (e.g., a single coded genome), because reading files in parallel would mostly increase the time for reading the files without contributing to evaluate the Target FS. Extending the benchmark for parallel tasks is left as future work.

5.3.2 Implementation and Usage

The FS-BioBench was implemented using the Python programming language and will be publicly released as a free and open-source software under the GNU general public license (GPL) version 3.0. It still have two dependencies that we intend to reduce in future versions: the Python Imaging Library (PIL) and large input files. The dependency on the PIL exist because we use this library to create an image file for the Manhattan plot from GWAS workflows. However, we can create a synthetic file with the same size by writing a simple binary file. The dependency on large input files can be avoided if we specify that users should first run the workflows `W1.Genotyping` and `W2.Sequencing` before running the others. The only workflow that will depend on files that FS-BioBench still does not write is the `W3.Prospection`, which needs the MIABIS files in the XML format, but these files are small and can be provided together with the benchmark code.

Currently, the basic usage steps of FS-BioBench are:

1. Download and extract the benchmark code in the local FS (see Figure 5.1).
2. Download, copy to the target FS and extract the input files.
3. Run the workflows.

Users only need to execute the following Python command to run the workflows:

```
$ python workflows.py <the_workflow> [<arg1> <arg2> ...]
```

In the following, we present the different available workflows (parameter `<the_workflow>`) and their expected arguments (parameter `[<arg1> <arg2> ...]`):

```
w1 <num_variations> <out_genotyping>
```

where `<num_variations>` is the number of variations to be written in the `<out_genotyping>` file.

```
w2 <num_reads> <read_size> <out_fastq>
```

where `<num_read>` is the number of NGS reads of size `<read_size>` to be written in the `<out_fastq>` file.

```
w3 <in_biobanks> <in_collections> <out_result>
```

where `<in_biobanks>` is the MIABIS XML file containing the list of biobanks, `<in_collections>` is the MIABIS XML file containing the list of samples from those biobanks, and the `<out_result>` file is the list of prospected samples.

```
w4 <in_fastq> <out_sam>
```

where `<in_fastq>` is the input FASTQ file, and `<out_sam>` is the output SAM file containing all aligned reads.

```
w5 <in_fastq> <out_fasta>
```

where `<in_fastq>` is the input FASTQ file, and `<out_fasta>` is the output FASTA file containing the contiguous genome sequence obtained from assembling all the reads.

```
w6 <in_genotyping_1> <in_genotyping_N> <n_individuals_vcf>
    <out_vcf> <out_odds> <out_plot>
```

where `<in_genotyping_1>` is the first genotyping input file, `<in_genotyping_N>` is the last genotyping input file, and the `<n_individuals_vcf>` is the number of individuals to be added between the first and the last individuals. The `<out_vcf>`, `<out_odds>`, and `<out_plot>` are the output VCF, odds and Manhattan plot files.

```
w7 <in_fastq> <out_sam> <max_num_lines_vcf>
    <out_vcf> <out_annotated>
```

where `<in_fastq>` is the input FASTQ file, and `<out_sam>` is the output SAM file containing the alignment of all reads. The `<max_num_lines_vcf>` is the maximal number of lines to be written in the VCF file, while the `<out_vcf>` and `<out_annotated>` are the output VCF and annotated VCF files respectively.

```
w8 <in_fastq> <out_sam> <max_num_lines_bed>
    <out_bed> <max_num_lines_diff> <out_diff>
```

where <in_fastq> is the input FASTQ file, and <out_sam> is the output SAM file containing the alignment of all reads. The <max_num_lines_bed> is the maximal number of lines to be written in the BED file, and the <out_bed> is the output BED file. The <max_num_lines_diff> is the maximal number of lines in the diff file, and <out_diff> is the output diff file.

5.3.3 Experiments

This section contains a set of experiments exemplifying the utility of our macrobenchmark, the FS-BioBench. They evaluate local file systems, network file systems, and different configurations and optimizations within the same file system. The computational environment is comprised of two physical machines equipped with two quad-core processors (Intel Xeon E5520), 32GB of RAM memory and one hard disk with 146 GB (SCSI, 15k RPM). One of the two machines was used to evaluate the local file systems and as a client machine when evaluating the network file systems. The other machine was used as a network file system server only.

In `W1.Genotyping`, we set the benchmark to create a genotyping file with 960614 entries (knwon SNPs). This number comes from the 23AndMe genotyping files [1], for example, those freely available in the Personal Genome Project. `W2.Sequencing` writes 4715311 FASTQ entries with sequences with length of 100 base pairs, which results in approximately 1GB. `W6.GWAS` creates the intermediate VCF file with 100 individuals. `W7.Annotation` writes a VCF file with 5.6M genomic variations. `W8.Methylation` estimates the genotype and methylation status of 960000 positions in the intermediate file and writes 18000 methylated candidates in the final output file.

Comparing Local File Systems

A local file system abstracts the hardware device to allow users read and write files instead of data blocks [102]. The performance of this component is crucial for the overall user experience in computers since several programs need to read files from or write to a disk. In this experiment, we compare some existent file systems from GNU/Linux and Windows, namely: `ext2`, `ext3`, `ext4`, `btrfs`, and `fat32`. Figure 5.2 contains the resulting values from FS-BioBench workflows running on these systems.

The more modern features a file system offers, the better is its usability for bioinformatics and several other use cases. However, complex features come with a cost. For example, the `fat32`, `ext2` and `ext3` file systems have less features than the `ext4`, but they still perform better than the latter in almost all workflows. The only exception is `btrfs`, which is the most modern file system from the evaluated ones, and performs well in all workflows, however it still incurs in some outliers that need to be better studied.

Comparing Network File Systems

Network file systems transparently provide common file system operations that are executed in a remote file server [104]. The network becomes a member of the entire system since it transports requests and data between clients and servers. NFS and SSHFS are two examples of this type of systems, and follow different design principles and security premises. The former implements a protocol proposed by Sun Microsystems in 1984 (now, v4.1), which is largely used in small

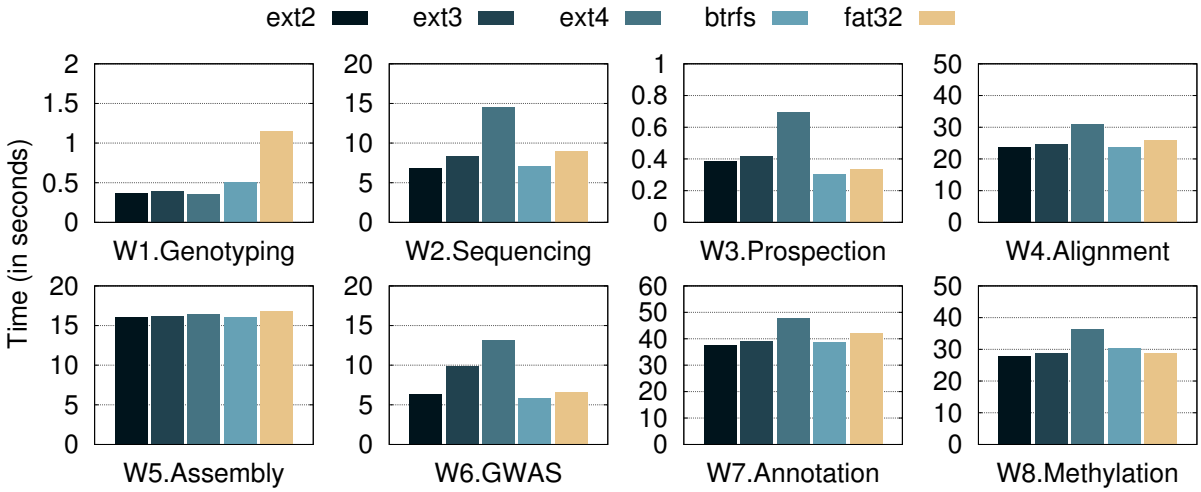


Figure 5.2: Results from running the FS-BioBench workflows in local file systems.

LAN clusters and traditionally has some security limitations in its default mode. The later is a client program that uses the SSH File Transfer Protocol (SFTP) with an original Secure Shell (SSH) server running anywhere, which means it is a secure solution with its additional incurred cost. Some argue comparing NFS and SSHFS is the same as comparing apples to oranges. However, people compare them in practice for deciding up to which point the security brought by SSHFS compensate the performance overhead. Figure 5.3 presents the results from this comparison with focus on showing the performance from the user perspective.

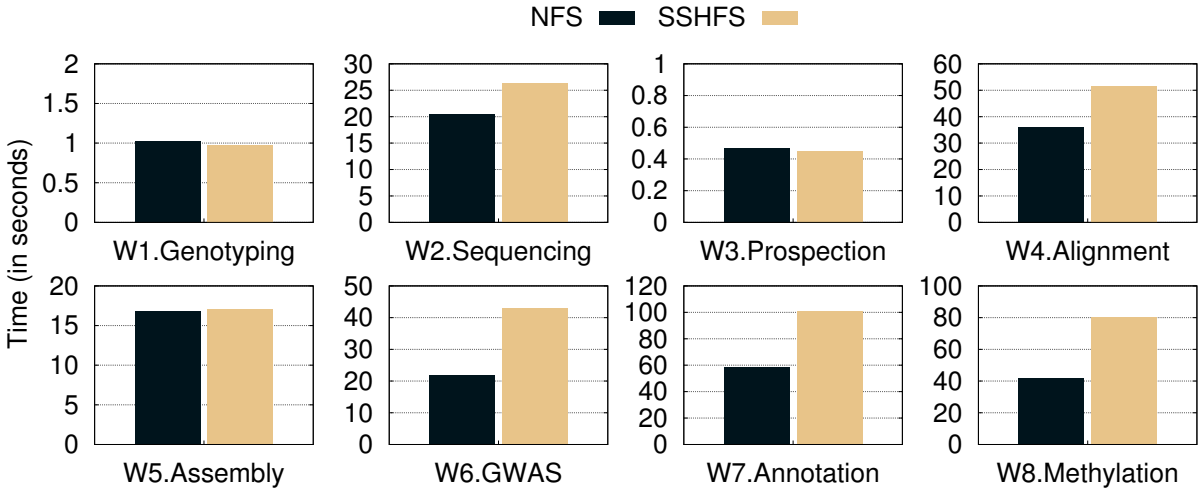


Figure 5.3: Results from running the FS-BioBench workflows in cluster file systems.

As expected, SSHFS is only as fast as NFS in some workflows, and its overhead may reach 100% in others (e.g., in `W8.Methylation`). However, we intend to show with this experiment that our benchmark can help system administrators to decide which solution has the best

cost-benefit for their environments, even when the overall comparative result is known beforehand.

Evaluating Configurations and Optimizations of SSHFS

Every file system provides different internal configurations and optimizations that might interfere with the overall system's performance. For example, synchronous write operations determines if each to-be-written data is immediately stored in the persistent media or if it is stored in batches from time to time, which defines also the consistency and durability guarantees provided by the file system. Caching and prefetching data are also noteworthy functionalities among several file systems. A cache avoids data transfers when reading files that already were read before and still reside in the cache system. Prefetching is a proactive mechanism that tries to bring data blocks from files being read sequentially before they are actually requested. System administrators may execute a macrobenchmark to obtain the best parameters for a specific file system that maximize its utility.

We compare some of the mentioned aspects using the SSHFS in this experiment, namely: cache, prefetching and asynchronous writes. Figure 5.4 contains the results from this test. We used the option `-o ServerAliveInterval=30` in all experiments, which activates a heartbeat mechanism to maintain active the connection between the client and the server. The *opt* case includes the SSHFS' default configuration, which activates the cache, the prefetching and asynchronous writes. The *no-cache* case removes only the cache from the *opt* case, with the option `-o cache=no`. The *no-pref* case removes only the prefetching operation from the *opt* case, with the option `-o no_readahead`. The *sync-w* case makes all write operations synchronous with the option `-o sshfs_sync`. Finally, the *no-opt* case applies the three previous configurations together, which removes the cache, the prefetching and write data synchronously.

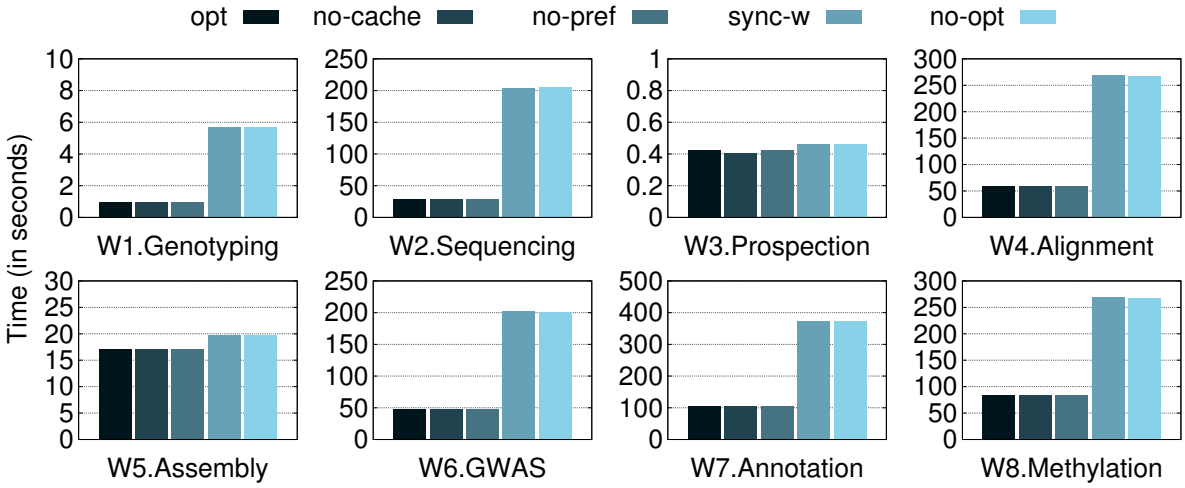


Figure 5.4: Results from running the FS-BioBench workflows over SSHFS with and without optimizations. The y-axes from this image are all in logarithmic scale.

The overhead caused by writing data synchronously (*sync-w* case) is the most outstanding result from this figure. As expected, the more write-intensive the workflow is, and the bigger the data is, the higher this overhead is. In these cases, the *sync-w* case takes 2 to 6× longer

than the *opt* case to finish each FS-BioBench workflow. It happens because every entry (e.g., a sequencing read) is sent to and stored in the network file system server before returning to the next write operation on the client side.

The results from disabling the cache (*no-cache*) and prefetching (*no-pref*) optimizations showed no significant overhead compared to the default configuration of SSHFS. One should perform a deeper analysis on some specific aspects to better understand and control their impact in the results. For example, the default space dedicated to the cache may be insufficient for maintaining most blocks of files already read by our workflows. Additionally, the default size and number of data blocks prefetched by the client-side of SSHFS file system may also be insufficient for improving the read performance.

5.3.4 Discussion

Macrobenchmarks have at least two known limitations: representativeness and isolation [102, 103]. The first limitation is related with the fact that the tool may be unrepresentative for other workflows beyond those implemented in the benchmark. First, users should attest the chosen benchmark is broad enough to synthesize the common tasks in their areas. Second, writing new workflows to the benchmark is the only available solution to broaden it to the missing areas. The other limitation, isolation, means that running this type of benchmark may be insufficient to isolate a specific metric or the influence of a single optimization. The solution encompasses the complementary use of microbenchmarks for that.[1]

Additionally, macrobenchmark users must carefully address two potential pitfalls: results with a high standard deviation and the cache warm-up [102, 103]. Network transfers and disk accesses may result in high standard deviation values. The less multi-tenancy the system has, the lower the standard deviation is. Possible solutions to this problem include guaranteeing the benchmark is running alone in the system, and executing the benchmark more times to reduce the impact of outliers. The cache warm-up is the other pitfall, which happens when the benchmark requests cause several cache miss at the beginning of its execution, until some cache hits are achieved. It may misguide the results' interpretation because there are two separated phases: the warm-up and the dirty cache, which cannot be considered as one. The solution relies in cleaning the cache before each benchmark execution, or waiting the warm-up to finish before start logging the results (if it is the case).

5.4 Related Work

Other bioinformatics benchmarks can be divided into two categories: macrobenchmarks covering a wide range of tasks, and competition benchmarks focusing on a single task. Both groups measure the overall performance of a computing system, which include storage, network, and processing resources.

BioBench [38], BioPerf [43], BioInfoMark [79], and MineBench [91] are examples from the first type. They focus mainly on searching and aligning small sequences, and executing proteomics, phylogenomics and genetics workflows. These topics use broadly accepted tools (e.g., BLAST, PHYLIP, HMMER), and thus most of these benchmarks rely and depend on external tools to be executed. In FS-BioBench, we use bioinformatics workflows from NGS life-cycle, simulate the I/O from representative tasks on these workflows, and try to avoid as much as

[1] As we did when evaluating CHARON (see Section 2.6).

possible any dependency in complex processing tools. The reasoning for this is that if we employed external tools, we would have to support updates to our benchmark when new versions of these tools are released. Additionally, benchmark users have to install several external tools from different sources before start running the experiments. MineBench is the only mentioned benchmark that still is actively maintained by their creators, while the others were discontinued in 2005 (BioBench and BioPerf) or 2008 (BioInfoMark).

CASP [90], GAGE [97] and Assemblathon [52] are examples from the second type. They intend to compare and rank the algorithmic performance of several solutions for the same task. CASP is a competition for protein structure prediction. GAGE and Assemblathon are competitions for genome assembly algorithms. These competition benchmarks measure two main metrics: the quality of results and the speed to find a solution. In FS-BioBench, we focus on measuring the performance of simulated I/O operations based on representative bioinformatics tasks.

5.5 Final Considerations

In this chapter, we presented a file system macrobenchmark called FS-BioBench, which is based on bioinformatics workflows and on the interactions between researchers and data repositories. We discussed the workflows considered in the FS-BioBench by classifying the main processes involving these two entities into five categories. The proposed tool was implemented in Python and will be publicly released as a free and open-source software. Our experiments demonstrate the utility of our solution to compare local or network file systems, or even to evaluate the effects of different configurations and optimizations within the same file system.

Chapter 6

RANC

Chapter Authors:
Ricardo Fonseca and Nuno Neves and Alysson Bessani (FFCUL).

6.1 Introduction

The Resilient and Adaptive Network Communication (RANC) channel is an interconnected set of processes that work cooperatively to provide a robust, fault-tolerant and secure communication to applications. The RANC is a follow-up to the overlay network Resilient Event Bus (REB), which was developed within the European project MASSIF[1]. The RANC inherits most of its features from REB, which are briefly enumerated here (for more details, refer to [59]):

- Peers in the RANC, denominated as RANC nodes, receive data from applications in the form of a stream of bytes, and transmit those bytes via point-to-point communication.
- RANC nodes authenticate themselves before and during transmission of data to provide authenticity and integrity of data; confidentiality is also provided on an optional basis.
- RANC nodes transmit data over an underlying UDP/IP network, and can use multiple paths in the physical and overlay networks to increase spatial redundancy for communication resiliency (overlay paths have at most one intermediary node).
- Erasure coding is applied over the transmitted data to reduce the cost of replication by senders and also increase the recovery efficiency by receivers.
- Communication paths are periodically probed by source nodes to select the fastest and most reliable links during data transmission.

In CHARON, we plan to use RANC to transfer files directly between BiobankCloud instances when such files are located in private repositories. In the next section, we outline the general techniques employed in RANC that make it an interesting substitute for the secure TCP channels we employ for direct file transfer on CHARON.

[1]MASSIF (MAnagement of Security information and events in Service InFrastructures) is a collaborative project co-funded under the European Commission's FP7 ICT Work Programme 2009 (FP7-ICT-2009-5). It is aligned with the objective ICT-5-1.4 - Trustworthy ICT. Project reference: FP7-257475. Website: http://www.massif-project.eu/

6.2 Communication Properties

RANC nodes communicate through the UDP/IP protocols, defining an overlay network atop the IP network and running application-level routing strategies to select overlay channels that are (expectedly) providing correct communication. Overlay networks have been used as mechanisms to implement routing schemes that take into account specific application requirements [40]. In RANC's scenario, we employ an overlay network to create redundant network-agnostic channels for efficient and robust communication.

Point-to-point communication channels between nodes is one of the features supported by RANC. Data is transmitted as a stream of bytes, and is delivered reliably in a first-in first-out (FIFO) order. The arrival of duplicate data is identified and removed, and flow control is enforced at the senders to prevent receivers from being overwhelmed with too much information. Additionally, the RANC ensures data integrity, authenticity and confidentiality.

RANC also provides a robust multipath communication that tolerates some faults in the underlying network and also in intermediary RANC nodes, which would interrupt the communication if, for instance, a single TLS/TCP channel was used instead. The usage of multiple paths to transmit data from a source to a destination is done in an efficient way by taking advantage of erasure codes to minimize retransmissions, and which is further optimized by picking the paths with the best quality of service.

In the remaining of this section, we generically explain how these properties are achieved.

6.2.1 Reliable Delivery

Reliable delivery of data can be attained when the underlying communication network provides at least fair loss delivery, that is, if it makes some effort to transmit a packet through a route and then deliver it to the destination. The Internet is one of such networks, and using UDP on top of it does not strengthen its fair loss property, therefore it may happen that a transmitted packet is lost in-transit (e.g., due to congestion or crash of a network router). On the other hand, TCP provides reliable delivery (assuming both source and destination processes are correct) by segmenting the input stream and by transmitting one segment at a time, while waiting for the arrival of an acknowledgment of its reception. A retransmission occurs if apparently the segment was lost, e.g., when the retransmission timer expires at the sender. As each segment is received and acknowledged, the receiver delivers it to the application.

RANC employs a similar strategy to provide reliable delivery. The input stream from the application is saved in a queue, and then split into several segments for dissemination. Unlike TCP, however, retransmissions based on timers are avoided when possible because we employ spatial redundancy (transmitting through multiple routes) to provide the necessary robustness. To minimize the message overhead of the redundancy, the RANC also pre-processes each segment with an encoder that applies an erasure code – a kind of Forward Error Correction (FEC) code – to produce a number of packets. Depending on the code that is applied, if it is systematic or not, the resulting packets may contain the original data plus some repair information, or they may just have encoded data. The overall sum of the packets lengths is typically larger than the original segment, but it becomes feasible to reconstruct the segment even if some of the packets are lost.

The sender RANC node disseminates the packets as they are produced by the encoder, and starts a timer that should expire case retransmissions have to be performed, however this should only happen rarely for two reasons: (1) the erasure codes can recover from the loss of a subset of the transmitted packets; (2) a route monitoring mechanism is used to select those paths

that expectedly have the highest quality of service. The receiver RANC node accumulates the arriving packets in a receive queue, and when enough packets of a given segment are available, the receiver attempts to decode them. In case of success, it returns an acknowledgement back to the sender. Otherwise, it waits for the arrival of an extra packet for this segment before trying again to decode.

If the retransmission timer at the sender does indeed expire, then based on the latest information received from the acknowledgement, the sender node retransmits any unacknowledged packets (in this case the timer is reinitialized with a larger value). This process is repeated until the recovery of the original segment is accomplished. Depending on the network conditions, packets/segments may arrive and be decoded out of order. To address this issue, the RANC utilizes a selective acknowledgement scheme to convey to the source information about which packets/segments have already arrived.

6.2.2 Ordered and Duplication-free Delivery

A fair loss network like the Internet does not ensure an ordered delivery of packets. This occurs because different packets, sent by one source, may experience distinct delays when transmitted through diverse routes. Additionally, there is the possibility of spurious transmission of duplicate packets, which often happens due to re-routing algorithms in intermediate nodes. Despite these difficulties, TCP provides FIFO ordering at the delivery and also removes duplicates. This is accomplished by assigning to each segment a sequence number, which is used on the receiver side to order the segments. Furthermore, the sequence numbers are used to detect and discard duplicate segments.

In the RANC, the transmission of a segment corresponds to the dissemination of a fixed number of packets, which encode part of the original segment data and some redundant bytes. Each segment has an associated sequence number that is incremented monotonically. These sequence numbers are employed to keep track of lost segments and to detect data duplication (accidental or from replay attacks). Packets also have a distinct sequence number, which increases monotonically per segment. Therefore, a packet is unequivocally identified by carrying in the header a pair composed of the sequence number of the segment it belongs to plus its own sequence number. Packets that arrive with the same identifiers are detected and removed as duplicates.

Within the same segment, packets that arrive out of order according to their sequence number are normally accepted by the decoding process, so unordered reception per segment is tolerated. FIFO ordering is enforced with the segment sequence number, and a receiver node can deliver data in the right order by buffering complete unordered segments until all their predecessors have been given to the application. The amount of unordered data that is maintained in a RANC node is managed through a receiver sliding window flow control mechanism. This mechanism prevents the receiver from accumulating too much unordered data, something that could be used by a malicious RANC node to overflow the memory of the receiver. The sender is informed of how much empty space is still available inside the window, and packets that arrive beyond this space are discarded.

6.2.3 Authentication, Data Integrity and Confidentiality

Unreliable networks cannot be trusted to guarantee security properties such as the authenticity, integrity or confidentiality of transmitted data from a source to a destination. An attacker may eavesdrop the communication to access sensitive data if she can intrude any intermediate node

in the network path. Additionally, if the attacker has sufficient privileges, she can also alter transmitted packets or introduce new ones in the communication to perform attacks such as intrusions at the endpoints or man-in-the-middle impersonations.

Secure protocols have been devised throughout the years to provide the necessary protection to communications over unreliable networks, for example some at the session level (such as SSL/TLS), and others at the network level (such as IPSec). Cryptographic mechanisms exist for guaranteeing the necessary security properties of data transmission, and these are usually employed at the communication endpoints but also at the intermediate nodes in some cases. The RANC operates in a distributed environment where the nodes cooperate to increase the robustness of the communication by making use of multiple overlay paths to add spatial redundancy. Therefore, the RANC needs to guarantee security properties on its own with the robustness aspect in mind.

At configuration time, a secret cryptographic key is assigned to every pair of RANC nodes, which is shared exclusively by the two of them. This key is used to protect the communication from attacks, supporting the authentication of nodes and the integrity/authenticity of the data, and (optionally) its confidentiality. Every time a communication session between a pair of nodes is established, two cryptographic keys are created based on the shared secret key. One of the session keys is used to provide confidentiality of the data, where the content of a transmitted packet is encrypted at the sender and decrypted at the receiver, using the key. The other key is used to authenticate the nodes and provides the integrity/authenticity of the data.

Every transmitted packet has a Message Authentication Code (MAC) appended, which is generated using the content of the packet and the key. MACs are verified at the receiver before packet delivery, and packets are discarded if their MACs do not match the expected values. Since transmitted data is divided into segments, which are subdivided into multiple packets, we could have chosen to alternatively append a MAC to the whole segment and verify it only after the full reception. This solution would have the virtue of saving some MAC calculations, both at the sender and receiver, whenever segments are large. However, it suffers from a few drawbacks, making it less appealing in practice. First, the verification would be postponed, allowing corrupted packets to occupy space on the receiver buffers until much later. Second, the receiver node cannot separate good from bad packets, and therefore, a single damaged packet would cause the whole segment to be dropped (and then later retransmitted again).

RANC nodes transmit packets using multiple concurrent routes, which can be based on a single direct channel between the source and the destination, or can have a channel from the source to an intermediary node and then another channel to the final receiver. Assuming that the sender and the receiver are both correct, then attacks can occur both on the network and on the intermediary node (in case this node was compromised). On direct channels, only one MAC is generated per packet by the source, using the authentication key shared with the destination node. On two-hop channels, a pair of MACs is created, the first for the intermediary node and the second for the destination node. After receiving a packet, the intermediary validates and removes its MAC and only then forwards the packet to the destination. Here, again, one could save a few MAC calculations if a single MAC was added independently of the type of route (i.e., direct or two-hop). However, since MACs are obtained relatively fast, we decided that it was better to have the capability to immediately identify and delete modified packets, both at the intermediary and final nodes. This feature is also helpful to determine if certain channels are under attack, since we can pinpoint with good precision where the fault occurred.

6.2.4 Robustness Through Multipath

The overlay network created among the RANC nodes allows for multiple distinct routes (or paths) to be taken to transmit data to a specific destination. The source can, for instance, send the data directly or ask one of the other RANC nodes to forward it to the destination. It is expected that these two paths will go through distinct physical links, and therefore, localized failures will only disrupt part of the communication. Based on this insight, the RANC uses multipath communication to send data concurrently over several different routes in the overlay network. These routes consist either of direct paths between the source and destination nodes, or paths in which an intermediary node receives data from a source and redirects it to the destination. The RANC resorts to a one-hop source routing scheme, in which the overlay route of each message is defined at the sender (source routing), based on the local knowledge of the state of the links, and is composed of at most one intermediate relaying RANC node (one-hop).

Given a certain segment m, the RANC could send a copy of it over k distinct paths. This would allow $k-1$ path failures to be tolerated, at the cost of the transmission of $k-1$ extra segment replicas – the overhead is $(k-1) \times m$. Depending on the segment size, this cost can be significant especially when the network is behaving correctly (which is the expected normal case). Additionally, it also requires the receiver to process and discard $k-1$ duplicates. To address this difficulty, the RANC employs erasure coding techniques to ensure that packet loss can be recovered at the receiver, at a much lower replication overhead. Basically, the sender needs to do some processing on the segment m to produce some extra repair information r, and then $m+r$ is divided into several packets that are transmitted over various links. Even if only a subset of the segment and repair data arrives, the receiver can still recover the original segment. RaptorQ [82] is the selected scheme for erasure coding due to its tiny overhead of repair information and encoding/decoding efficiency.

A sender RANC node also periodically probes each of its most promising routes to a given destination to derive a quality metric to be associated with the path. Based on this metric, the sender can determine at each moment which are the best routes for a destination, and select them to disseminate a segment. Since the metric is calculated based on actual data collected from the network, it may require some period of time for the value of the metric to adjust after sudden changes in the network. During this period the sender could still think that a specific path is good, and consequently continue to use the path for transmissions, when in fact most packets could be lost. Moreover, a malicious intermediary RANC node could attack the measurement process, for instance by making its paths look particularly good, and then suddenly start dropping all packets. In this scenario, the usage of erasure codes helps minimizing the impact of sudden changes in a route's quality, because if one assumes that the failure only affects a limited number of routes, the remaining ones still deliver enough packets for the original segment to be reconstructed.

6.3 A Note on Software-Defined Networks

In deliverable D4.1 [49], we pointed out that software-defined networks (SDNs) can bring interesting benefits in terms of network utilization when used to transfer and disseminate the huge amount of data handled by a federation of biobanks [49]. One of the motivations for considering this within the context of the project was the results recently shown by Google, which improved their network utilization dramatically by employing an SDN control plane to manage the traffic on their inter-datacenter backbone [74]. Given the potentially huge amount of data

that can be transferred in a biobank federation [111], we believe the same benefits obtained by Google could be achieved in our scenario.

During the first two years of BiobankCloud, the FFCUL team did some foundational work for implementing a distributed control plane for software-defined networks [51, 50], aiming to enable a scenario in which the control of the network will be distributed among the members of the federation (and not centralized, as in the Google case). Our initial idea was to integrate such distributed network control plane with the capabilities of the RANC middleware, to improve channel failure recovery and flow control.

However, after some analysis of the current state of biobank integration and the objectives of work package 4, we decided to stop working on SDN and concentrate our effort in implementing the CHARON data-sharing capabilities. The first reason for that is the dependency of SDN technology on the deployment of modern switches that support the OpenFlow control protocol [89]. Currently, most biobanks and bioinformatics research institutions do not support this technology. The second reason for dismissing further work on SDNs in the BiobankCloud project is to focus all our efforts (i.e., PMs) in realizing our vision of a secure, decentralized Dropbox-like service for science in CHARON.

We still believe SDNs may bring important benefits for the integration of Biobanks (and the work done in biobank can be part of that), but we think these benefits should be exploited in further projects, which consider the design of a network backbone interconnecting these institutions.

Bibliography

[1] 23AndMe – genetic kit for ancestry. dna service. https://www.23andme.com/.

[2] Amazon DynamoDB – NoSQL cloud database service. http://aws.amazon.com/dynamodb/.

[3] Amazon S3. http://aws.amazon.com/s3/.

[4] Amazon S3 access control list (ACL) overview. http://docs.aws.amazon.com/AmazonS3/latest/dev/ACLOverview.html.

[5] Amazon S3 pricing. http://aws.amazon.com/s3/pricing/.

[6] BaseSpace. https://basespace.illumina.com/.

[7] Box. https://www.box.com/.

[8] Dropbox number of users announcement. http://techcrunch.com/2014/04/09/dropbox-hits-275m-users-and-launches-business-product-to-all/.

[9] Filebench webpage. http://sourceforge.net/apps/mediawiki/filebench/.

[10] Google cloud datastore – NoSQL database for cloud data storage. https://cloud.google.com/datastore/.

[11] Google cloud storage access control. http://developers.google.com/storage/docs/accesscontrol.

[12] Google drive. https://drive.google.com/.

[13] Google storage. https://developers.google.com/storage/.

[14] Google storage pricing. https://developers.google.com/storage/docs/pricingandterms.

[15] HP public cloud. http://www.hpcloud.com/products-services/storage-cdn.

[16] HP public cloud documentation. https://docs.hpcloud.com/api/object-storage#general_acls-jumplink-span.

[17] Lunacloud. http://www.lunacloud.com/pt/cloud-storage.

[18] Lunacloud predefined permissions documentation. http://www.lunacloud.com/docs/tech/storage-restful-api.pdf.

[19] Message queuing service with simple API – Rackspace cloud queues. http://www.rackspace.com/cloud/queues/.

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

[21] Microsoft onedrive. https://onedrive.live.com/about/pt-br/.

[22] Rackspace ACLs documentation. http://www.rackspace.com/blog/create-cloud-files-container-level-access-control-policies/.

[23] Rackspace cloud files. http://www.rackspace.co.uk/cloud/files.

[24] Rackspace cloud files pricing. http://www.rackspace.com/cloud/files/pricing/.

[25] Rackspace temporary urls documentation. http://docs.rackspace.com/files/api/v1/cf-devguide/content/TempURL-d1a4450.htmll.

[26] S3FS - FUSE-based file system backed by Amazon S3. http://code.google.com/p/s3fs/.

[27] S3QL - a full-featured file system for online data storage. http://code.google.com/p/s3ql/.

[28] The Galaxy Project – Online bioinformatics analysis for everyone. http://galaxyproject.org/.

[29] Ubuntu one. https://one.ubuntu.com/.

[30] Windows Azure. http://www.windowsazure.com/pt-br/solutions/storage-backup-recovery/.

[31] Windows Azure permissions documentation. http://msdn.microsoft.com/en-us/library/windowsazure/dn140255.aspx.

[32] Windows Azure predefined permissions documentation. http://msdn.microsoft.com/en-us/library/windowsazure/dd179354.aspx.

[33] Windows Azure pricing. http://www.windowsazure.com/en-us/pricing/details/.

[34] Windows Azure shared access signature documentation. http://azure.microsoft.com/en-us/documentation/articles/storage-dotnet-shared-access-signature-part-1/.

[35] Ittai Abraham, Gregory Chockler, Idit Keidar, and Dahlia Malkhi. Byzantine disk paxos: optimal resilience with byzantine shared memory. *Distributed Computing*, 18(5):387–408, 2006.

[36] Hussam Abu-Libdeh, Lonnie Princehouse, and Hakim Weatherspoon. RACS: A case for cloud storage diversity. *SoCC*, 2010.

[37] A. Adya et al. Farsite: Federated, available, and reliable storage for an incompletely trusted environment. In *OSDI*, 2002.

[38] Kursad Albayraktaroglu, Aamer Jaleel, Xue Wu, Manoj Franklin, Bruce Jacob, C-W Tseng, and Donald Yeung. Biobench: A benchmark suite of bioinformatics applications. In *Performance Analysis of Systems and Software, 2005. ISPASS 2005. IEEE International Symposium on*, pages 2–9. IEEE, 2005.

[39] Naomi Allen et al. UK Biobank: Current status and what it means for epidemiology. *Health Policy and Technology*, 1(3):123–126, 2012.

[40] D. Andersen, H. Balakrishnan, F. Kaashoek, and R. Morris. Resilient overlay networks. In *Proceedings of the ACM Symposium on Operating Systems Principles*, pages 131–145, 2001.

[41] Thomas E. Anderson, Michael D. Dahlin, Jeanna M. Neefe, David A. Patterson, Drew S. Roselli, and Randolph Y. Wang. Serverless network file systems. *ACM Trans. on Computer Systems*, 14(1):41–79, February 1996.

[42] Hagit Attiya and Jennifer Welch. *Distributed Computing: Fundamentals, Simulations and Advanced Topics*. Wiley Series on Parallel and Distributed Computing. Wiley-Interscience, 2nd edition, 2004.

[43] David A Bader, Yue Li, Tao Li, and Vipin Sachdeva. Bioperf: A benchmark suite to evaluate high-performance computer architecture on bioinformatics applications. In *Workload Characterization Symposium, 2005. Proceedings of the IEEE International*, pages 163–173. IEEE, 2005.

[44] Monya Baker. Next-generation sequencing: adjusting to data overload. *Nature Methods*, 7(7), June 2010.

[45] Luiz André Barroso, Jimmy Clidaras, and Urs Hölzle. *The datacenter as a computer: An introduction to the design of warehouse-scale machines, 2nd edition*. Synthesis lectures on computer architecture. Morgan & Claypool Publishers, 2013.

[46] C. Basescu et al. Robust data sharing with key-value stores. In *DSN*, 2012.

[47] Alysson Bessani, Miguel Correia, Bruno Quaresma, Fernando Andre, and Paulo Sousa. DepSky: Dependable and secure storage in cloud-of-clouds. *ACM Transactions on Storage*, 9(4), 2013.

[48] Alysson Bessani, Ricardo Mendes, Tiago Oliveira, Nuno Neves, Miguel Correia, Marcelo Pasin, and Paulo Verissimo. SCFS: a shared cloud-backed file system. In *Proc. of the 2014 USENIX ATC*, 2014.

[49] A. Bessani et al. State of the art and preliminary architecture, November 2013. Deliverable D4.1 of BiobankCloud project.

[50] Fábio Botelho, Alysson Bessani, Fernando Ramos, and Paulo Ferreira. On the design of practical fault-tolerant SDN controllers. In *Proc. of the 3rd European Workshop on Software Defined Networks – EWSDN'14*, September 2014.

[51] Fábio Botelho, Fernando Ramos, Diego Kreutz, and Alysson Bessani. On the feasibility of a consistent and fault-tolerant data store for SDNs. In *Proc. of the 2nd European Workshop on Software Defined Networks – EWSDN'13*, October 2013.

[52] Keith R Bradnam, Joseph N Fass, Anton Alexandrov, Paul Baranay, Michael Bechner, Inanç Birol, Sébastien Boisvert, Jarrod A Chapman, Guillaume Chapuis, Rayan Chikhi, et al. Assemblathon 2: evaluating de novo methods of genome assembly in three vertebrate species. *GigaScience*, 2(1):1–31, 2013.

[53] Chris Bryant. European data protection under a cloud. http://www.ft.com/cms/s/0/dbee868a-f43c-11e2-8459-00144feabdc0.html#axzz2gyIqMlYl.

[54] Paul R Burton, Anna L Hansell, Isabel Fortier, Teri A Manolio, Muin J Khoury, Julian Little, and Paul Elliott. Size matters: just how big is big? quantifying realistic sample size requirements for human genome epidemiology. *International Journal of Epidemiology*, 38(1):263–273, 2009.

[55] M. Cahill, U. Röhm, and A. Fekete. Serializable isolation for snapshot databases. In *Proceedings of the 2008 ACM SIGMOD international conference on Management of data - SIGMOD '08*, pages 729–738, 2008.

[56] B. Calder et al. Windows Azure storage: a highly available cloud storage service with strong consistency. In *SOSP*, 2011.

[57] Gregory Chockler and Dahlia Malkhi. Light-weight leases for storage-centric coordination. *International Journal of Parallel Programming*, 34(2), April 2006.

[58] Murim Choi et al. Genetic diagnosis by whole exome capture and massively parallel DNA sequencing. *Proc. of the National Academy of Sciences*, 106(45):19096–19101, 2009.

[59] MASSIF Consortium. Deliverable D5.1.4: Resilient SIEM Framework Architecture, Services and Protocols. Project MASSIF EC FP7-257475, September 2013.

[60] Lin Dai, Xin Gao, Yan Guo, Jingfa Xiao, Zhang Zhang, et al. Bioinformatics clouds for big data manipulation. *Biology direct*, 7(1):43, 2012.

[61] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. Dynamo: Amazon's highly available key-value store. In *Proc. of 21st ACM SIGOPS Symposium on Operating Systems Principles - SOSP'07*, 2007.

[62] M. A. C. Dekker. Critical Cloud Computing: A CIIP perspective on cloud computing services (v1.0). Technical report, European Network and Information Security Agency (ENISA), December 2012.

[63] I. Drago et al. Inside Dropbox: Understanding personal cloud storage services. In *IMC*, 2012.

[64] Paul Flicek and Ewan Birney. Sense from sequence reads: methods for alignment and assembly. *Nature methods*, 6:S6–S12, 2009.

[65] G. Gibson et al. A cost-effective, high-bandwidth storage architecture. In *ASPLOS*, 1998.

[66] C. Gray and D. Cheriton. Leases: An efficient fault-tolerant mechanism for distributed file cache consistency. In *Proc. of the 12th ACM Symposium on Operating Systems Principles – SOSP'89*, 1989.

[67] Ayat Hatem, Doruk Bozdağ, Amanda E Toland, and Ümit V Çatalyürek. Benchmarking short sequence mapping tools. *BMC bioinformatics*, 14(1):184, 2013.

[68] David Haussler et al. A million cancer genome warehouse. Technical report, University of Berkley, Dept. of Electrical Engineering and Computer Science, 2012.

[69] Maurice Herlihy. Wait-free synchronization. *ACM Trans. Programing Languages and Systems*, 13(1):124–149, 1991.

[70] Holger Heyn et al. Distinct DNA methylomes of newborns and centenarians. *Proc. of the National Academy of Sciences*, 109(26):10522–10527, 2012.

[71] J. Howard et al. Scale and performance in a distributed file system. *ACM Trans. Computer Systems*, 6(1):51–81, 1988.

[72] Michael Hummel, Lora Dimitrova, Karin Zimmermann, Jrgen Brandt, and Ulf Leser. Identification of high density data and use cases, November 2013. Deliverable D6.2 of BiobankCloud project.

[73] Raj Jain. *The art of computer systems performance analysis*. John Wiley & Sons, 1991.

[74] Sushant Jain, Alok Kumar, Subhasree Mandal, Joon Ong, Leon Poutievski, Arjun Singh, Subbaiah Venkata, Jim Wanderer, Junlan Zhou, Min Zhu, Jon Zolla, Urs Hölzle, Stephen Stuart, and Amin Vahdat. B4: experience with a globally-deployed software defined wan. In *Proceedings of the ACM SIGCOMM*, SIGCOMM '13, 2013.

[75] J. Kaye, H. Gottweis, F. Bignami, E. Rial-Sebbag, R. Lattanzi, and M. Macek Jr. Biobanks for Europe: A challenge for governance. Technical report, European Commission, Directorate-General for Research and Innovation, 2012.

[76] J.J. Kistler and M. Satyanarayanan. Disconnected operation in the Coda file system. *ACM Trans. Computer Systems*, 10(1):3–25, 1992.

[77] J. Kubiatowicz et al. Oceanstore: An architecture for global-scale persistent storage. In *ASPLOS*, 2000.

[78] Heng Li and Nils Homer. A survey of sequence alignment algorithms for next-generation sequencing. *Briefings in bioinformatics*, 11(5):473–483, 2010.

[79] Yue Li, Tao Li, Tamer Kahveci, and José Fortes. Workload characterization of bioinformatics applications. In *Modeling, Analysis, and Simulation of Computer and Telecommunication Systems, 2005. 13th IEEE International Symposium on*, pages 15–22. IEEE, 2005.

[80] Jan-Eric Litton, Roxana Merino Martinez, Jane Reichel, Karin Zimmermann, Lora Dimitrova, and Michael Hummel. Object model for biobank data sharing, June 2013. Deliverable D1.2 of BiobankCloud project.

[81] Rafael Los, Dave Shacklenford, and Bryan Sullivan. The notorious nine: Cloud Computing Top Threats in 2013. Technical report, Cloud Security Alliance (CSA), February 2013.

[82] M. Luby, A. Shokrollahi, M. Watson, T. Stockhammer, and L. Minder. RaptorQ Forward Error Correction Scheme for Object Delivery. IETF RFC 6330, August 2011.

[83] Daniel G MacArthur et al. A systematic survey of loss-of-function variants in human protein-coding genes. *Science*, 335(6070):823–828, 2012.

[84] D. Malkhi and M.K. Reiter. Secure and scalable replication in phalanx. In *Proc. Seventeenth IEEE Symposium on Reliable Distributed Systems – SRDS'98*, Oct 1998.

[85] Dahlia Malkhi and Michael Reiter. Byzantine quorum systems. *Distributed Computing*, 11(4):203–213, 1998.

[86] Elaine R Mardis. The impact of next-generation sequencing technology on genetics. *Trends in genetics*, 24(3):133–141, 2008.

[87] Ali José Mashtizadeh, Andrea Bittau, Yifeng Frank Huang, and David Mazières. Replication, history, and grafting in the ori file system. In *Proc. of the 24th ACM Symposium on Operating Systems Principles*, 2013.

[88] Mark I McCarthy, Gonçalo R Abecasis, Lon R Cardon, David B Goldstein, Julian Little, John PA Ioannidis, and Joel N Hirschhorn. Genome-wide association studies for complex traits: consensus, uncertainty and challenges. *Nature Reviews Genetics*, 9(5):356–369, 2008.

[89] Nick McKeown, Tom Anderson, Hari Balakrishnan, Guru Parulkar, Larry Peterson, Jennifer Rexford, Scott Shenker, and Jonathan Turner. OpenFlow: enabling innovation in campus networks. *SIGCOMM Comput. Commun. Rev.*, 38(2):69–74, March 2008.

[90] John Moult. A decade of casp: progress, bottlenecks and prognosis in protein structure prediction. *Current opinion in structural biology*, 15(3):285–289, 2005.

[91] Ramanathan Narayanan, Berkin Ozisikyilmaz, Joseph Zambreno, Gokhan Memik, and Alok Choudhary. Minebench: A benchmark suite for data mining workloads. In *Workload Characterization, 2006 IEEE International Symposium on*, pages 182–188. IEEE, 2006.

[92] Loreana Norlin, Martin N Fransson, Mikael Eriksson, Roxana Merino-Martinez, Maria Anderberg, Sanela Kurtovic, and Jan-Eric Litton. A minimum data set for sharing biobank samples, information, and data: MIABIS. *Biopreservation and biobanking*, 10(4):343–348, 2012.

[93] Tiago Oliveira, Ricardo Mendes, and Alysson Bessani. Sharing files using cloud storage services. In *Second Workshop on Dependability and Interoperability in Heterogeneous Clouds (DIHC), co-alocated with Euro-Par*, 2014.

[94] Jane Reichel, Roxana Martinez, and Jan-Eric Litton. Draft report of legal and ethical framework and an analysis of ethical viability, March 2013. Deliverable D1.5 of BiobankCloud project.

[95] R. L. Rivest, A. Shamir, and L. Adleman. A method for obtaining digital signatures and public-key cryptosystems. *ACM Trans. Computer Systems*, 21:120–126, 1978.

[96] Heli Salminen-Mankonen, Jan-Eric Litton, Erik Bongcam-Rudloff, Kurt Zatloukal, and Eero Vuorio. BBMRI–the Pan-European research infrastructure for biobanking and biomolecular resources: managing resources for the future of biomedical research. *EMBnet. news*, 15(2):pp–3, 2009.

[97] Steven L Salzberg et al. GAGE: A critical evaluation of genome assemblies and assembly algorithms. *Genome research*, 22(3):557–567, 2012.

[98] Michael C Schatz, Ben Langmead, and Steven L Salzberg. Cloud computing and the DNA data race. *Nature biotechnology*, 28(7):691, 2010.

[99] Emil Stefanov, Marten van Dijk, Ari Juels, and Alina Oprea. Iris: a scalable cloud file system with efficient integrity checks. In *ACSAC*, 2012.

[100] Lincoln D Stein et al. The case for cloud computing in genome informatics. *Genome Biol*, 11(5):207, 2010.

[101] J. Stribling et al. Flexible, wide-area storage for distributed system with WheelFS. In *NSDI*, 2009.

[102] V. Tarasov, S. Bhanage, E. Zadok, and M. Seltzer. Benchmarking file system benchmarking: It *IS* rocket science. In *HotOS*, 2011.

[103] A. Traeger, N. Joukov, C. P. Wright, and E. Zadok. A nine year study of file system and storage benchmarking. *ACM Trans. on Storage*, 4(2):25–80, May 2008.

[104] Paulo Verissimo and Luis Rodrigues. *Distributed Systems for System Architects*. Kluwer Academic Publishers, January 2001.

[105] Paulo Esteves Verissimo and Alysson Bessani. E-biobanking: What have you done to my cell samples? *Security Privacy, IEEE*, 11(6):62–65, 2013.

[106] Michael Vrable, Stefan Savage, and Geoffrey M. Voelker. BlueSky: A cloud-backed file system for the enterprise. In *FAST*, 2012.

[107] David G Wang et al. Large-scale identification, mapping, and genotyping of single-nucleotide polymorphisms in the human genome. *Science*, 280(5366):1077–1082, 1998.

[108] R William G Watson, Elaine W Kay, and David Smith. Integrating biobanks: addressing the practical and ethical issues to deliver a valuable tool for cancer research. *Nature Reviews Cancer*, 10(9):646–651, 2010.

[109] David A Wheeler et al. The complete genome of an individual by massively parallel DNA sequencing. *Nature*, 452(7189):872–876, 2008.

[110] H-Erich Wichmann et al. Comprehensive catalog of European biobanks. *Nature biotechnology*, 29(9):795–797, 2011.

[111] C. Wilks et al. The cancer genomics hub (CGHub): overcoming cancer through the power of torrential data. *The Journal of Biological Databases and Curation*, 2014.

[112] Zhe Wu, Michael Butkiewicz, Dorian Perkins, Ethan Katz-Bassett, and Harsha V Madhyastha. Spanstore: Cost-effective geo-replicated storage spanning multiple cloud services. In *Proc. of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, pages 292–308. ACM, 2013.

[113] Martin Yuille et al. Biobanking for Europe. *Briefings in bioinformatics*, 9(1):14–24, 2008.

[114] Daniel R Zerbino and Ewan Birney. Velvet: algorithms for de novo short read assembly using de Bruijn graphs. *Genome research*, 18(5):821–829, 2008.