BIOBANKCLOUD
Your PaaS for Biobanking

SEVENTH FRAMEWORK PROGRAMME

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

D4.3
Overbank Implementation and Evaluation

Due date: 31st May, 2015 (M30)
Deliverable version: 1.0

Editor
Alysson Bessani (FFCUL)

Contributors
Ricardo Mendes, Tiago Oliveira, Vinicius Cogo, Alysson Bessani (FFCUL)

Contents

Document History

Version	Date	Description	Authors	Reviewers
0.1	2015-05-01	First draft.	Alysson Bessani Ricardo Mendes Vinicius Cogo Tiago Oliveira	
1.0	2015-05-29	Final version.	Alysson Bessani Ricardo Mendes Vinicius Cogo Tiago Oliveira	

Chapter 1

Introduction

The CHARON file system is the implementation of the *Overbank* concept from the BiobankCloud project, which aims to integrate storage repositories from several biobanks transparently, and let them leverage the virtually infinite storage capacity from public cloud providers. This document contains an overview of the CHARON prototype, as well as how to install and configure it. The produced software is available in a ZIP file, which accompanies this document or can be downloaded from `https://docs.google.com/uc?id=0B3HxGhFWPkMKeWVsdlBfV2hnTzg`.

Efficiency and security are prime concerns for all storage systems from the BiobankCloud project. A detailed description of CHARON architecture, its design choices, implementation, and evaluation are available in the Deliverable D4.2 [4]. In the appendix of this document we present an updated version of this description, showing several important performance improvements when compared with the version we had last November. Additionally, the Deliverable D2.4 [5] presents the steps required to integrate CHARON with Hops-FS, the other storage system from the project, which focuses on managing and processing data from a single biobank.

The next chapters from the present deliverable contain an overview of CHARON prototype (Chapter 2), and describe how to configure and use it (Chapter 3). In Appendix A we present an updated version of the description and evaluation of Charon (initially presented in Deliverable D4.2 [4]).

Chapter 2

The CHARON file system

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

CHARON is a cloud-backed file system capable of storing and sharing big data in a secure, reliable, and efficient way using multiple cloud providers and storage repositories. It is secure and reliable because it does not require trust on any single entity, and it supports the storage of different types of data in distinct locations to comply with required privacy premises. Two distinguishing features of CHARON are its serverless design (no client-managed server is required in the cloud) and its efficient management of large files (by employing prefetching, cache, and background writes). The complete description of CHARON architecture and its internal protocols are available in the Deliverable D4.2 of the BiobankCloud project [4].

The distributed nature of CHARON allows users to store data in diverse locations to comply with different privacy requirements [6, 7]. Three types of data locations are supported in CHARON: cloud-of-clouds, single (public) storage cloud, and private repository. The cloud-of-clouds provides the availability of multi-clouds scenarios [3] and confidentiality, but has a 50% increased cost. The single storage cloud requires trust in a provider, but is less expensive than the previous case. Private repositories provide a dependability level that depends on the adopted techniques and solutions, but normally incurs in scalability limitations. The first solution may be used to store research biological data in a secure way, the second to store public consent data, and the third to store clinical data. Independently on the location of data, CHARON allows users to share it.

Figure A.1 illustrates a scenario where two biobanks store their data in local repositories, in single public cloud providers, and in a resilient cloud-of-clouds. In this scenario, the namespace tree has six nodes: directories `d1` and `d2`, and files `A`, `B`, `C`, and `D`. The namespace is maintained in the cloud-of-clouds, together with file `B`. File `D`, less critical, is kept in a single cloud. File `A` is stored locally because it cannot leave Biobank 2. File `C` is shared between the two sites (e.g., in the same country), thus being stored in both of them.

As in many other distributed file systems, CHARON separates file data and metadata (e.g., file name and permissions) in distinct objects that are stored in diverse locations and are managed using different strategies. CHARON stores the metadata in a cloud-of-clouds always, using the protocols from DepSky [3], which guarantee the privacy of metadata by encrypting it at the client side. All the metadata is stored within namespace objects, which encapsulate the hierarchical structure of the files and directories in a subdirectory tree. CHARON maintains the namespace tree, together with the files' metadata, replicated in the cloud-of-clouds storage to avoid a single point of failure and clients loosing data if they loose their local namespace tree.

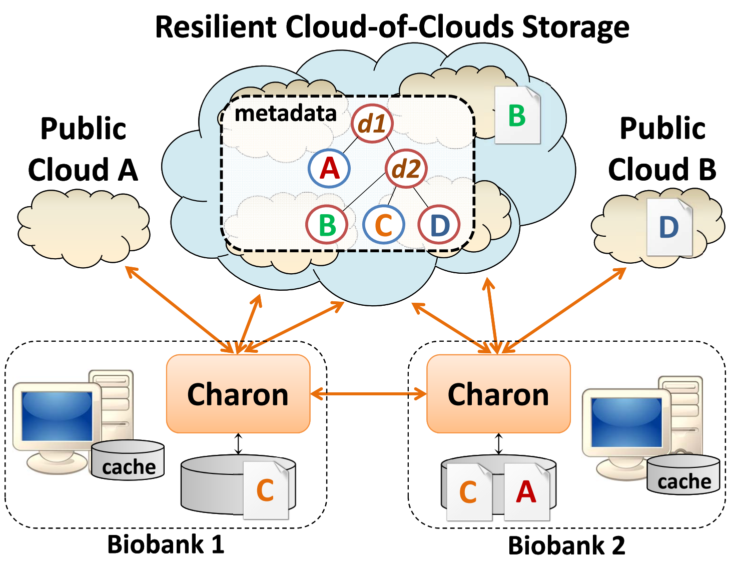


Figure 2.1: CHARON overview.

A data-centric Byzantine-resilient leasing algorithm is used in CHARON to avoid concurrency conflicts when clients share portions of their namespaces.

Several techniques for efficient dealing with big data without requiring custom services on the cloud make CHARON a promising solution. A multi-level cache reduces the need of always having to fetch data from their data locations, which reduces the monetary costs on the traffic required to download data from the clouds. It is multi-level because it uses the local disk to cache the most recent files used by clients, and it uses a fixed small main memory cache to improve subsequent data access over open files. CHARON splits files into fixed-size blocks to fit data in the memory cache and allow users to start processing data as soon as the first blocks are read, instead of waiting to end the download of the whole file. Since files are split in blocks, CHARON is able to prefetch sequential blocks as soon as a sequential read is detected, which reduces the time required to read a file. Finally, as mentioned before, CHARON is able to store data in diverse data locations according with different privacy requirements.

CHARON provides a POSIX interface for users, and it was implemented in Java as a file system in user space (FUSE). It means that users interact with CHARON (after installed and configured) in the same way they interact with any other file system (e.g., `etx4`). The system is fully implemented at client side, using cloud services for storage and coordination. Currently, CHARON connects with four public cloud providers: Amazon S3, Windows Azure Storage, Rackspace Cloud Files, and Google Cloud Storage.

CHARON will perform the inter-datacenter tasks and the storage processes between biobanks and public clouds in the BiobankCloud platform. On the other hand, Hops-FS has a view of all data being stored in a single datacenter and efficiently processes it. In the next section, we present the step-by-step tutorial on how to install, configure, and use CHARON.

Chapter 3

Configuring and using CHARON

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

In this chapter, we present a tutorial with the steps performed to configure and deploy CHARON. Additionally, we also discuss how users interact with the tools and commands to share data using this storage solution from the BiobankCloud project.

3.1 Prerequisites

Before configuring and deploying CHARON, we assume users have accomplished some prerequisites in an GNU/Linux environment (we tested it in Ubuntu 12.04 and in newer versions). First, CHARON has software dependencies on the Filesystem in Userspace (FUSE) and the Java platform (version 1.7 or newer). They need to be installed and ready to be executed in your operating system.

Second, users must sign up in the cloud providers and obtain the API keys, which will be used by CHARON to store the system data and metadata. Currently, we support four cloud storage providers as storage backend, which are:

- **Amazon S3** (`http://aws.amazon.com/s3/`)
- **Google Storage** (`https://developers.google.com/storage/`)
- **RackSpace** (`http://www.rackspace.co.uk/`)
- **Windows Azure** (`https://www.windowsazure.com/en-us/`)

After creating the user accounts in the cloud providers, you need to obtain the API keys that allow CHARON to access them.

To get the API key from **Amazon S3**, go to the “AWS Management Console”, click in “S3 service”, click in your account name (in the upper right corner), and go to the “Security Credentials”. After that, in the Access Keys separator you can generate your access and secret keys.

In the case of **Google Storage**, go to the “Google API Console”, go to “Google Cloud Storage” separator, choose “Interoperable Access”, and there you can find the API key. Do not forget to enable Google Cloud Storage in the services separator first.

The **RackSpace** API key is accessible by going to its "Control Panel", and you will find how to get your secret key in the upper right corner. The access key is the username you use in this account.

To find the API key in **Windows Azure**, go to your account portal, create a new storage project, select it, and you can find the key management at the bottom of the page. In this case, your access key is the name of your storage project and you secret key is the primary key in the key management.

3.2 Configuring CHARON

The first two steps encompass downloading the latest available version of CHARON[1] and extracting it.

```
wget http://link-download-charon.com
tar -xvzf charon.zip
```

Before configuring CHARON, you must link the system with the FUSE, which can be done by executing a script provided in the root directory of the extracted project:

```
./configure_libjavafs.sh
```

Set values to the variables `JAVA_HOME` and `FUSE_HOME` in the `build.conf` file (also in the project's root folder). Usually, the `FUSE_HOME` is the `/usr/local` directory. The `JAVA_HOME` depends on the Java version installed on your machine.

```
sed -i 's/'JAVA_HOME='/'JAVA_HOME=/x/y/z'/g'
sed -i 's/'FUSE_HOME='/'FUSE_HOME=/usr/local'/g'
```

The next step is to set the configuration files for the data locations of interest. Since you have the API keys from cloud storage providers, you are able to define the configuration of the system's locations.

Cloud-of-clouds. To define the configuration of this location you need to fill up the `config/CoC.properties` with the API keys from the cloud storage providers. An example of this file using the four cloud storage providers we support is presented bellow:

[1] *http://link-download-charon.com*

```
driver.type=AMAZON-S3
driver.id=cloud1
accessKey=********************
secretKey=****************************************
location=EU_Ireland

driver.type=GOOGLE-STORAGE
driver.id=cloud2
accessKey=********************
secretKey=****************************************

driver.type=RACKSPACE
driver.id=cloud3
accessKey=********************
secretKey=****************************************

driver.type=WINDOWS-AZURE
driver.id=cloud4
accessKey=********************
secretKey=****************************************
```

If you only want to use Amazon S3 in a cloud-of-clouds environment (using different availability zones) as your cloud storage provider, you can only create an account at this cloud provider and use the example file provided (`config/CoC_amazon.properties`). To do that, copy the content of the "`CoC_amazon.properties`" file to the one mentioned before (`CoC.properties`). In this case, four different Amazon S3 locations will be used to store the data (`US_Standard`, `EU_Ireland`, `US_West` and `AP_Tokyo`).

Single-cloud. The file you must set with the credentials of the cloud storage provider used to store the data of files of this location is the `config/singleCloud.properties` file. Here is an example of this file configured to use Amazon S3 as storage backend:

```
driver.type=AMAZON-S3
driver.id=cloud1
accessKey=********************
secretKey=****************************************
location=EU_Ireland
```

Hops-FS Private Repository. To use Hops-FS as a CHARON private repository you need to set the `config/hopsfsRep.properties` file with the path of the directory to be used for storing the data inside Hops-FS. In order to use a directory called "charon-files" to do that, the content of this file must look like this:

```
hopsfs:/charon-files
```

As you can see, this materializes the Scenario 1 defined in Section **??**.

Before mounting the system, there are some system parameters you still must define and some others you can set according with your needs. Table 3.1 shows these parameters, explains their purpose, and defines which ones are mandatory. These parameters can be set in the `config/charon.config` file.

Parameter	Type	Description
`client.id` *	Integer	Unique identifier of the system.
`mount.point` *	String	The path to the directory in which Charon will be mounted.
`email` *	String	The email address associated with the client.
`addr` *	IP:Port	IP and Port to listen for private repository data requests.
`prefetching`	Boolean	Defines if the system uses prefetching. *(default: true)*
`compression`	Boolean	Defines if the system uses data compression. *(default: true)*
`num.backgroud.threads`	Integer	The max number of threads used to parallelly write the files' data. *(default: 4)*
`num.prefetching.threads`	Integer	The number of prefetching threads. *(default: 4)*
`num.depsky.data.threads`	Integer	Number of threads used by DepSky to read/write the filesystem's data. *(default: 12)*
`num.depsky.metadata.threads`	Integer	Number of threads used by DepSky to read/write the filesystem's metadata. *(default: 4)*
`share.tokens.directory`	String	Share tokens directory. *(default: ${project-directory}/shareTokens).*
`cache.directory`	String	The path to the cache directory *(default: ${project-directory}/cache_${client.id})*
`local.repository.directory`	String	The local repository address. Can be both a local and a remote folder. *(default: ${cache.directory}/privateRep)*
`personal.namespace.id`	String	Personal namespace identifier. If it is not set, it is filled up with a randomly generated one when the system is mounted.

Table 3.1: Charon parameters description. Properties marked with * are required.

3.3 Mounting CHARON

After configuring CHARON, you are able to mount it. To do that, you can use a script we provide by executing the following command in the project's root directory:

```
./Charon_mount.sh
```

At this point, the system will be mounted and accessible at the mount point defined at the `mount.point` field in the `charon.config` file.

3.4 Managing external files

CHARON is now able to manage files/directories that are external to the system. Users only need to inform CHARON that some resource should be managed by it. The system will immediately create the metadata for this resource and will periodically verify if there were modifications on it. One design choice that appears here is that these kind of resources are set as read-only inside CHARON making the system just a sharing agent for them. Given this, to add an external managed folder to CHARON, one only needs to execute the following command in the project's root folder:

```
./charon.manage <resource-to-be-managed-path> <charon-containing-folder>
```

In this command, the `resource-to-be-managed-path` parameter is the path of the folder to be managed, and the `charon-containing-folder` is the name of this folder inside the CHARON mount point. For example, if we want the folder `/home/user/studies` to be managed by CHARON in such a way that its content will appear in it as a folder called `external/studies`, the command would looks like the following:

```
./charon.manage /home/user/studies external/studies
```

Note that we are also able to define a specific file to be managed. An example of this would be the addition of the `/home/user/studies/study1.doc` file to the external managed resources of CHARON in the same directory of the previous example. In this case, the command would look like:

```
./charon.manage /home/user/studies/study1.doc external/studies
```

Finally, in the case of Hops-FS resources, they can be added to the CHARON external managed resources by executing the same command explained before adding the prefix `hopsfs:` to the path of the Hops-FS resource in the `resource-to-be-managed-path` parameter of the command. In this way, to make CHARON manage the Hops-FS's `studies` folder in a folder called `external/studies`, the command to be executed would look like:

```
./charon.manage hopsfs:/studies external/studies
```

As can be seen, this command materializes the Scenario 3 defined in Section **??**.

3.5 Sharing data

In order to share data, there are some information that must be exchanged between the CHARON instances. Figure 3.1 shows this process for two instances of the system deployed in two sites (*Site 1* and *Site 2* in the figure). The procedure is the following: in step 1, *Site 1* generates a `site_id` which contains the information other sites need to know to share data with him and send it to *Site 2* (step 2 in the figure). This token is a text file that is filled by the client with info about itself, the clouds it uses, and the site in which the system is deployed. After receiving the `site_id`, *Site 2* is able to share data with *Site 1*. Given this, when the former wants to share data with the latter it first generates a `share_token` and then send it to the *Site 1* (steps 3 and 4 in the figure). Contrary to the last token, this one is generated automatically by CHARON when a folder is shared. Finally, when an instance gets the `share_token` (step 5) it is able to access the shared data.

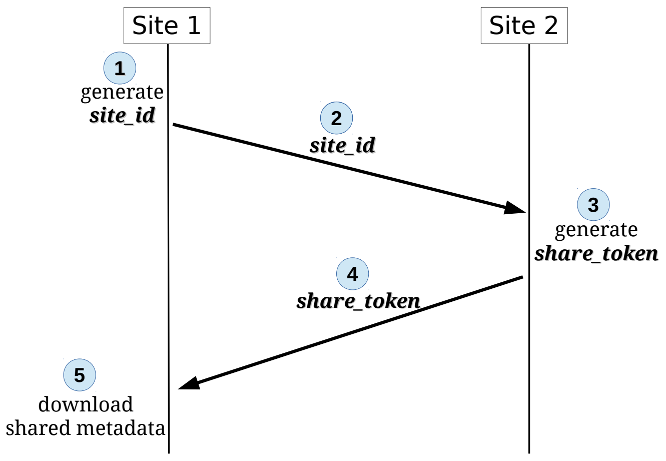


Figure 3.1: Exchange of tokens needed to share data.

Notice that steps 1 and 2 are executed only once when a CHARON instance introduces himself to other, and steps 3, 4 and 5 are executed every time some instance share a folder with other. The exchange of both `site_id` and `share_token` are not suppose to be done by the system. However, CHARON is able to perform this task using the users' e-mail. We are studying if there is some way to perform this task within the BiobankCloud LIMS.

As explained before, the `site_id` must be filled by the client. To do that, one just needs update the fields of the `/config/site.id` file. This file must look like:

```
id=4
email=user@gmail.pt
addr=127.0.0.1:11000

driver.type=AMAZON-S3
canonicalId=****************************************

driver.type=GOOGLE-STORAGE
canonicalId=user@gmail.com

driver.type=RACKSPACE
canonicalId=user,user@gmail.com
```

In this file, the fields must contain:

- **`id:`** The unique identifier of the CHARON instance. This id is defined in the `client.id` field in `config/charon.config` file.
- **`email:`** The email of the system client defined in the `config/charon.config` file.
- **`addr:`** IP and port where the system listens requests from the private repository, which are defined in `config/charon.config` file.
- **`driver.type`** and **`canonicalId:`** The `driver.type` identifies the cloud storage provider, and a `canonicalId` is used by other instances to give this client access to their data. Both must be defined for each cloud account used to store both filesystem's data and metadata.

The `canonicalId` format differ from cloud to cloud, which will be explained in the following descriptions together with an explanation on how to obtain them.

In the case of **Amazon S3**, in AWS control panel go to top left corner and click on your username and Security Credentials. Next, click on "Account Identifiers" and you will find the value of "Canonical User ID" there.

For **RackSpace**, the `canonicalId` is composed by the service account username and the client e-mail, separated by a comma. This e-mail could be the one defined in the `email` field on the top of the file.

In **Google Storage** this identifier is the Google account email.

Notice that there is no information about **MS Windows Azure**. This happens because in this service the technique used to give grantees access does not need any client specific information [4].

Given this, when a client wants to share a directory it can use a tool we provide in the project directory by executing the following command:

```
./charon.share <charon-dir> <peer-id> <permissions>
```

In this command the charon-dir parameter is the CHARON managed directory the client wants to share, the peer-id is the client with which the directory will be shared and the permissions define the capabilities the peer will have over that directory. In this way, to share a directory called studies/shared with the client with the id 5 giving him read-only permissions, the command to be executed should be this:

```
./charon.share studies/shared 5 r
```

The execution of this command generates the share_token for this shared folder and after that the resources inside the studies/shared folder will be accessible for the client 5. This share_token will be created in the directory defined by the share.tokens.directory field in the /config/charon.config file. As explained before, this token must be sent to the client with which the directory was shared (in our example client 5), and it must put it into a folder called NewSNSs in the project's root directory. The config/NewSNSs folder is monitored by CHARON in order to add new shared folders to the system.

3.6 Deploying CHARON with Chef/Karamel

The previous section has described how to manually configure and install CHARON. However, this process can be hard and confusing, specially for users that are non-computer experts. To facilitate the CHARON deployment, we provide a Chef cookbook [1] which atomically downloads and executes CHARON and its dependencies. Moreover, this cookbook can be used through Karamel [2] (which was developed under the BiobankCloud Project) to deploy CHARON together with other cluster software. Note that, although the CHARON Chef cookbook and the Karamel framework automate the installation of CHARON, the users still need to create the clouds accounts and find the required API keys, as explained in Section **??**.

CHARON Chef cookbook. For using the CHARON's cookbook the user must download it at GitHub in the following link: https://github.com/biobankcloud/charon-chef; extract it, and put it in a desired folder (e.g.: charon_cookbook). It provides two recipes: one for installing the software and all the dependencies (charon_cookbook/recipes/install.rb), and another for running the system (charon_cookbook/recipes/default.rb). For running this cookbook without using karamel (using Chef commands), the users must fulfill first the attributes in the Chef attributes file (which can be found in charon_cookbook/attributes/default.rb).

Using Karamel. The first thing to do is to download the last stable version of Karamel available on its web page [2]. After that, for deploying CHARON through Karamel the user only needs to download the "charon.yaml" file on the root of the CHARON's cookbook at GitHub. This file defines the cluster to be deployed, including the size of the cluster (1 machine by default), the cookbook GitHub link, the Amazon EC2 machine to be used (*m3.medium* by default) and the location of that machine (Ireland by default).

After running the Karamel framework, users will be able to load the "charon.yaml" file to Karamel interface. Before launching the cluster, the users must configure the CHARON attributes in that interface, for example the cloud account's API keys, the user email and the mount point of the filesystem. When launching the cluster, Karamel will first execute the recipe to install the software, and then the recipe to execute it.

Bibliography

[1] Chef. https://docs.chef.io/.

[2] Karamel. http://www.karamel.io/this.

[3] 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.

[4] A. Bessani et al. The overbank cloud architecture, protocols and middleware, November 2014. Deliverable D4.2 of BiobankCloud project.

[5] J. Dowling et al. Secure, scalable, highly-available file system integrated with the object model and overbank, May 2015. Deliverable D2.4 of BiobankCloud project.

[6] George Gaskell, Martin W Bauer, et al. *Genomics and Society:" Legal, Ethical and Social Dimensions"*. Routledge, 2013.

[7] 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.

Appendix A

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

The data flood coming from life science institutions 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 makes it difficult to use public clouds in this scenario. We present CHARON, a new cloud-backed file system capable of storing and sharing big data in a secure, reliable, and efficient way using multiple cloud providers and storage repositories. CHARON is secure and reliable because it does not require trust on any single entity, and it supports the storage of different types of data in distinct locations to comply with required privacy premises. Two distinguishing features of CHARON are its serverless design (no client-managed server is required in the cloud) and its efficient management of large files (by employing prefetching, cache, and background writes). Moreover, a conceptual contribution of the system is its novel fault-tolerant data-centric leasing protocol to avoid write-write conflicts between clients sharing files. We evaluate CHARON using microbenchmarks and application-based benchmarks simulating shared accesses and representative bioinformatics workflows. The results show that our unique design is not only feasible, but also presents better performance than alternative solutions.

A.1 Introduction

Attributes like cost-effectiveness, ease of use, and (almost) infinite scalability make the cloud a natural candidate to tackle the exponential growth of information being produced by life sciences institutions. Most of this data is being created by next generation sequencing technologies, which enable whole human genome sequencing for less than $1000 [27, 56], an essential premise for the emergence of advanced health care solutions (e.g., personalized medicine).

Unfortunately, many of these institutions are still reticent to adopt public cloud services. First, few bioinformatics tools and systems are already integrated with clouds, introducing difficulties to the researchers that are usually non-computer experts. Second, as with all organizations dealing with critical information, there are concerns about trusting data to externally-controlled services that occasionally suffer from unavailability and security incidents [38, 53, 35]. Finally, depending on the nature of the data being analyzed, there are legal restrictions impeding such institutions to outsource the storage and manipulation of some of the datasets [39, 41].

Biobanks are particularly susceptible to these concerns [47]. These institutions were originally designed to keep biological samples that could be later retrieved for research purposes. More recently, they are becoming responsible also for storing and analyzing the data related with such samples, a trend called e-biobanking [66]. A sequenced human genome can reach up to 300GB, and each individual can have his genome sequenced many times during his life. The problem is that *biobanks lack the scalable infrastructure for storing and managing this potentially vast volume of data*. Public cloud providers have plenty of resources for that.

Furthermore, the use of widely-accessible cloud services would facilitate the integration and sharing of data among biobanks, hospitals, and laboratories, serving as a managed repository for public and access-controlled datasets. This would enable research initiatives that are not possible today due to the lack of a sufficient number of samples in a single institution [32]. For example, 20–50k samples are required to study the interactions between genes, environment, and lifestyle that enables (or inhibits) a complex disease [68]. The rarer the disease is, the longer it takes to gather all necessary samples [32]. In many cases, given the rarity of some diseases, it is unlikely that a single hospital or research institute will ever be able to collect the required number of samples. The problem is *how to exploit the benefits of public clouds without endangering the security and dependability of biobank data.*

Motivated by these problems, we designed CHARON, a cloud-backed file system capable of storing and sharing big data with minimal management and no dedicated infrastructure. CHARON uses multi-cloud (or cloud-of-clouds) data replication [20, 29, 30] to avoid having any cloud service provider as a single point of failure, operating correctly even if a fraction of the providers are unavailable (which happens more often than one would expect [35]) or misbehave (due to bugs, intrusions or even malicious insiders – although rarer, such events do happened before [53]). Furthermore, to comply with data protection legislation, CHARON allows datasets to be stored in distinct locations (cloud-of-clouds, single cloud or private repository) for different tradeoffs on the guarantees and costs.

A distinguishing feature of CHARON is its *serverless design*, in the sense that it does not depend on any custom server running on the cloud, relying instead on cloud-managed services. This brings two important benefits for life science institutions (see §A.2). First, *cost savings* are expected since executing servers in cloud VMs is typically more expensive than resorting to cloud-managed services (e.g., Amazon S3 [2] or Azure Queue [10]). Second, a *significant reduction on management tasks* because calling cloud-managed services requires much less effort than keeping servers operating properly in VMs in the cloud [57].

Dropbox and similar systems do not solve the problems CHARON addresses since they (1) assume that all the stored and shared data fit in clients machines and (2) require absolute trust on a single provider/company. Similarly, other cloud-backed storage systems *have little or no support for big data and either do not offer any controlled sharing capabilities* [20, 15, 67, 62, 40] *or rely – fully or partially – on custom servers running on the cloud* [31, 71]. Previous serverless designs like ICStore [29] and DepSky [30] export only read/write registers (roughly equivalent to a disk block), which are substantially simpler than a fully-fledged big-data enabled distributed file system (see §A.7 for details).

CHARON employs a set of Byzantine-resilient data-centric algorithms [19, 30], including a novel leasing protocol to avoid write-write conflicts on files. Security is ensured without relying on any single entity for implementing access control, by exploring the access control capabilities of cloud services and storing the data using efficient secret sharing [50]. Furthermore, the system is capable of handling big data by dividing files in blocks (for better interaction with cloud services), employing erasure codes and compression (for storage-efficiency), and using prefetching and background uploads (for decreasing/hiding cloud-access latency). The way we integrate these techniques into a usable system makes the design of CHARON unique.

In summary, the paper contributions are:

- The design and implementation of CHARON, a new cloud-backed file system built to facilitate data sharing and storage, targeting the needs of biobanks and other life sciences research institutions (§A.3 and §A.5);
- The first Byzantine-resilient data-centric lease algorithm that exploits different cloud services currently available without requiring trust on any of them individually (§A.4);
- An extensive evaluation of CHARON, comparing it with several local, networked, and cloud-backed storage systems, using common microbenchmarks and a novel benchmark that captures the I/O of representative workflows employed by bioinformaticians (§A.6).

Although CHARON was motivated by life science challenges, the system can be used in any other domain where secure data sharing and storage is needed.

A.2 Life Science Data Sharing and Storage

CHARON design was driven by the need to support controlled data sharing and archival among life science institutions with minimal investments in infrastructure.

Sharing bioinfomatics data. Nowadays, most bioinformatics data sharing is based on public repositories [63]. This solution has significant limitations since relevant parts of the information cannot be made public (e.g., non-anonymized human genomes [39]), which calls for a more refined access model [66]. 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 define a set of minimal shareable metadata for biobank samples [59, 69], but still there is little support for sharing the data associated with them.

In the US there have been a few proposals to create large data warehouses for concentrating the storage and processing of genomic sequence information [44]. In such scenario, the main engineering concern is to scale the warehouse infrastructure to large data flows, something that can often be addressed with techniques similar to the ones employed in commercial internet-scale services [28].

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 [39, 41]. In fact, even US research centers will have to face these issues, for instance if they want to collaborate with foreigner counterparts or if individual states decide to approve 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 secure, flexible, legally-compliant, and cloud-based environment. This will enable research programs that otherwise would be hard or even impossible to establish due to the lack of sufficient number of samples [32].

Benefits and limitations of public clouds usage. Life sciences research institutions and biobanks are "perfect" customers for public clouds for several reasons. First, their core business is not to maintain large data storage and processing facilities, and thus they would be natural candidates to exploit the ease-of-use of public or community clouds [61]. This is attested, for instance, by the large cloud adoption in (non-privacy-sensitive) bioinformatics research in recent years [37]. Second, there is a flood of data currently being generated by next generation sequencing machines [27, 56] that needs to be stored, processed, and archived by such institutions. This demand calls for an almost infinite scalable storage, which can be 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, one would like 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 [26]. Cloud services would reduce substantially the maintenance effort, 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 [4] and Galaxy [17], 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 before, legal constraints and industry secrecy will always make it difficult for ubiquitous use of such platforms.

Our approach. In this paper we describe a system that supports data sharing among federated parties, and the 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.

A.3 CHARON Design

CHARON is a distributed file system that provides a POSIX interface to access an ecosystem of multiple cloud storage services and allows data transfer between clients. As in many other distributed file systems, it *separates file data and metadata* in distinct objects that are stored in diverse locations and are managed using different strategies.

Three types of data locations are supported in CHARON: cloud-of-clouds, single (public) storage cloud, and private repository (e.g., a private cloud). These alternatives explore various cost-dependability tradeoffs and address all placement requirements we have encountered with life sciences applications. For example, the cloud-of-clouds can be used to store critical data that needs the availability and confidentiality provided by the multi-cloud scenario (provider-fault tolerance and 50% increased cost [30]). A single storage cloud (provider-dependent and less expensive) could store non-critical public studies and anonymized datasets. Finally, private repositories (variable dependability and subject to local infrastructure restrictions) must be used to keep clinical data from human samples that cannot leave the boundaries of a particular biobank or country [39].

CHARON maintains the namespace tree, together with the files' metadata, replicated in the cloud-of-clouds storage. The rationale for this decision is to keep the file system structure

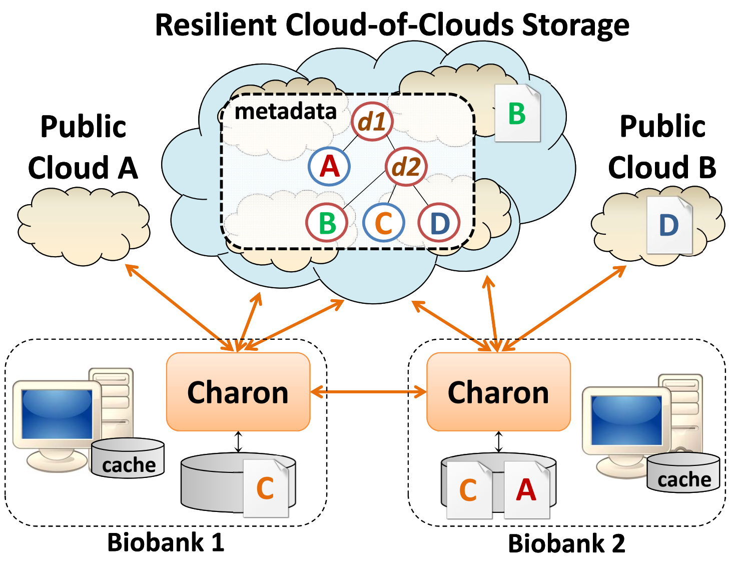


Figure A.1: CHARON architecture. The namespace tree has six nodes: directories `d1` and `d2`, and files `A`, `B`, `C`, and `D`. The namespace is maintained in the cloud-of-clouds, together with file `B`. File `D`, less critical, is kept in a single cloud. File `A` is stored locally because it cannot leave Biobank 2. File `C` is shared between the two sites (e.g., in the same country), thus being stored in both of them.

secure and available by exploiting (1) the expected high availability of cloud-of-clouds (which is greater than any local infrastructure or individual cloud) and (2) the efficient data-centric replication protocols developed in the last years [19, 30, 29]. The idea is to have only soft state in clients, which can be reconstructed after a crash by fetching data from other sources, and store all information about the system organization on the clouds. Figure A.1 illustrates the diverse storage locations in CHARON. Notice that independently on the location of the data, it is possible to share it.

Implementing this design implies addressing several challenges. In particular, the system needs to (1) efficiently deal with multiple storage locations, (2) manage reasonably big files, and (3) regulate concurrent access to shared files. These challenges are amplified by our goal of not having client-deployed servers (for decreasing costs and management effort) and requiring no modification on current cloud services (for immediate deployability). Addressing these challenges requires the right combination of several classical storage techniques (e.g., replication, erasure codes, prefetching), with recently proposed algorithms and concepts (e.g., DepSky' multi-cloud data replication [30], WheelFS' semantic cues [64]), and a novel data-centric protocol to coordinate writes in the system.

All these techniques were combined taking into consideration two important design decisions (besides avoiding trusting any individual cloud provider). First, CHARON absorbs file writes in the client local disk and then, in background, uploads it to its storage location. This choice aims to improve the usability of the system, since uploading genomic files (multiple gigabytes) to the clouds can take a significant amount of time (see §A.6).

The other design decision was to avoid write-write conflicts, ruling out any optimistic mechanism that relies on users/applications for conflict resolution [48, 58]. Again, the expected size of the files and the envisioned users justify this decision. More specifically, (1) solving conflicts manually in big files can be hard and time consuming, 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. This design decision avoids, for example, situations in which two users annotate the same shared genome file with medical information from different databases. In CHARON, when an honest client tries to open a file for writing, it first obtains a lease for the file, and only then updates it. An important contribution of our work is this leasing algorithm, which does not require trusting cloud providers or running custom servers. This will be described in the next section, while a description of other aspects of the system will be presented in §A.5.

A.4 Leases in the Cloud-of-Clouds

Leases are time-based contracts used to control concurrent accesses to resources (e.g., a file) while preventing version conflicts [43]. This section describes the novel Byzantine-resilient leasing algorithm used to avoid write-write conflicts in CHARON. All protocols described here are fully formalized and proved correct in a companion report [23].

A.4.1 Model and Guarantees

Our system model is equivalent to other models used in traditional data-centric Byzantine fault-tolerant algorithms (e.g., [19, 30]). An unbounded number of clients access a set of base objects (i.e., cloud services) that comprise the leasing service. Clients and up to f base objects can suffer arbitrary (or Byzantine) faults. A client may invoke, on the same base object, several parallel operations that are executed in FIFO order. Every base object provides access control to guarantee that only authorized clients invoke operations (see §A.5.3). Since the notion of lease implies timing guarantees, an upper bound is assumed for the message transmission time between clients and base objects. However, *this assumption is required only for liveness*.

Our algorithm ensures (always safe) mutual exclusion by allowing at most one process at time to access the shared resource [25] and only for a limited amount of time (the *lease term* [43]). Each resource provides three lease-related operations: `lease(T)` and `renew(T)` acquires and extends the lease for T seconds, respectively, while `release()` ends the lease. These operations satisfy the following properties:

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

These properties do not preclude a Byzantine client from acquiring a lease and renewing it indefinitely, nor prevent a direct access to the resource without a valid lease. This is acceptable because a malicious client can always damage all resources (i.e., files) he has access permissions. Additionally, if a client crashes holding a valid lease, the lease will be available again after at most T time units (time-boundedness property). We devised an algorithm satisfying only obstruction-freedom to pursue better performance in contention-free executions, since write-contention is expected to occur infrequently in the system.

A.4.2 Byzantine-resilient Composite Lease

Previous data-centric fault-tolerant mutual exclusion algorithms (e.g., [30]) are designed to work directly on top of storage services. In this paper we propose a more modular approach in which non-fault-tolerant *base lease objects* are build on top of a specific cloud-provided service, and $3f + 1$ of these services are combined in an f-fault-tolerant *composite lease object*. This approach allows the design of more efficient base lease objects on top of any cloud-provided service (e.g., queues) instead of relying on fault-tolerant register constructions as in [19, 34].

The lease operation of the composite lease object is presented in Algorithm 1. In order to acquire a composite lease, a client simultaneously calls the *lease* operation in each of the $3f + 1$ base lease object (Lines 5–6) and waits for either $2f + 1$ successes or $f + 1$ failures (Line 7). In the first case the client acquired the lease. Otherwise, the lease is unavailable or under contention and the client needs to release all potentially obtained leases – the successful and the unanswered ones (Lines 11–12) and backoffs, repeating the procedure after some time (Line 13). This algorithm is repeated until it succeeds or a timer expires (omitted for readability).

ALGORITHM 1: Composite resource leasing by client c.

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

Releasing a lease requires invoking the *release()* operation in all base objects (similarly to Lines 11-12). Renewing is similar to the lease algorithm, but with an additional cloud access to remove the information related with the lease being renewed. These two operations are never executed in the critical path of CHARON, and thus have no impact on the latency of the system.

A.4.3 Base Lease Implementations

Most public cloud services, from object storage to atomic database-as-a-service, provide basic features to create base lease objects. However, every implementation of a base lease object must comply with some specifications to work together in a composite lease. First, the `lease` operation requires the successful creation of (various flavors of) lease entries in the cloud service. Second, clients are responsible to garbage-collect their outdated or invalid lease entries to save resources. In most implementations, this cleanup requires at least one cloud access in the lease and renew operations. Third, lease entries must be signed before they are sent to the cloud to ensure that cloud providers cannot create or corrupt leases. Fourth, base lease implementations use the access control from cloud services to guarantee that only authorized clients can access a lease. In doing so, malicious clients can only hinder correct users that inadvertently gave them access to the resource. Fifth, clients do not assign local timestamps to the lease (to mark its starting period), instead they rely on the cloud service to add a timestamp on the lease entries or use the timestamps returned on every cloud invocation. These timestamps are also used to check lease validity (instead of local clocks). In the following, we describe four example base lease objects we implemented with different services offered by popular cloud providers.

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

This algorithm works in services like Amazon S3 [2], Google Storage [7], Azure Blob Storage [33], and Rackspace Files [13] since all of them guarantee strong consistency when creating objects and provide a timestamp on every service reply.

Augmented queues such as Windows Azure Queue [10] and Rackspace Queue [9] have strongly-consistent enqueue, dequeue, and list functions, providing thus an universal shared memory abstraction capable of solving synchronization problems [45].

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

NoSQL databases store data as pairs containing a unique key associated with a set of values. Amazon DynamoDB [1] provides a strongly-consistent service with a conditional update operation that enables the implementation of efficient base lease objects.

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

Transactional databases store data in tables and support ACID transactions. Google Datastore [6] is a cloud-based database-as-a-service that supports atomic transactions, which allows the implementation of base lease objects.

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

A.4.4 Comparing Base Lease Objects

Table A.1 summarizes the properties of the different base lease objects implemented on top of different cloud services.

This table shows that most base objects require three cloud accesses for implementing the *lease* operation. Obstruction-freedom is the progress property supported by all algorithms, and most algorithms that are not based on object storage satisfy also deadlock-freedom [25]. Deadlock-freedom guarantees that if two or more clients concurrently try to acquire a lease, one of them will succeed. However, our composite lease satisfies only the obstruction-freedom property, even if the n base objects satisfy deadlock-freedom.

There is a significant difference in the costs of running the base lease algorithms. However, for low-contention scenarios, our composite lease will be significantly cheaper than running a fault-tolerant lock service in multiple clouds VMs, especially if one considers that leases are needed only for writing and maintained for some time (see next section). More specifically, each lease acquisition would cost around $\mu\$40$, while having VMs in four different providers may cost up to $\$1200$ per month plus the traffic load (which is significant since the replicas need to synchronize on each lease operation) [31] and the management effort.

A.5 CHARON Implementation

CHARON was implemented in Java as a FUSE-based user-level file system. The system is fully implemented at client side, using cloud services for storage and coordination.

A.5.1 Metadata Organization

Metadata is the set of attributes assigned to a file/directory, including its name, location, parent directory, time of creation, permissions, etc. Independently of the location of the data blocks (e.g., a private repository), CHARON stores all metadata in the cloud-of-clouds using the DepSky' *single-writer multi-reader register implementation* (see §A.5.2). The rationale for this decision is to improve the accessibility and availability guarantees achieved by a multi-cloud setup, as no single cloud has perfect availability [35] and there are no reports of multi-cloud unavailability/security events. Furthermore, privacy is not endangered because metadata is encrypted before being stored.

Namespaces management. All the metadata is stored within namespace objects, which encapsulate the hierarchical structure of the files and directories in a subdirectory tree. CHARON uses two types of namespaces: *personal namespace* (PNS) and *shared namespace* (SNS). A PNS stores the metadata for all non-shared objects of a client, i.e., files and directories that can

Service	Accesses	Costs ($\mu\$$)	Progress
Amazon S3	3	15	Obst.-Free
Google Storage	3	30	Obst.-Free
Azure Blob Storage	3	0.108	Obst.-Free
RackSpace Files	3	8.640	Obst.-Free
Azure Queue	3	0.15	Dead.-Free
RackSpace Queue	3	30.144	Dead.-Free
Amazon DynamoDB	2	subscription*	Dead.-Free
Google Datastore	4	1.2	Obst.-Free

Table A.1: Base lease objects built on top of cloud services. The table shows the number of cloud accesses required to acquire a lease in absence of contention; the monetary costs (in microdollars) of such operation; the progress property satisfied by each base lease object algorithm, either obstruction-freedom or deadlock-freedom (which is stronger). *This service is charged per month.

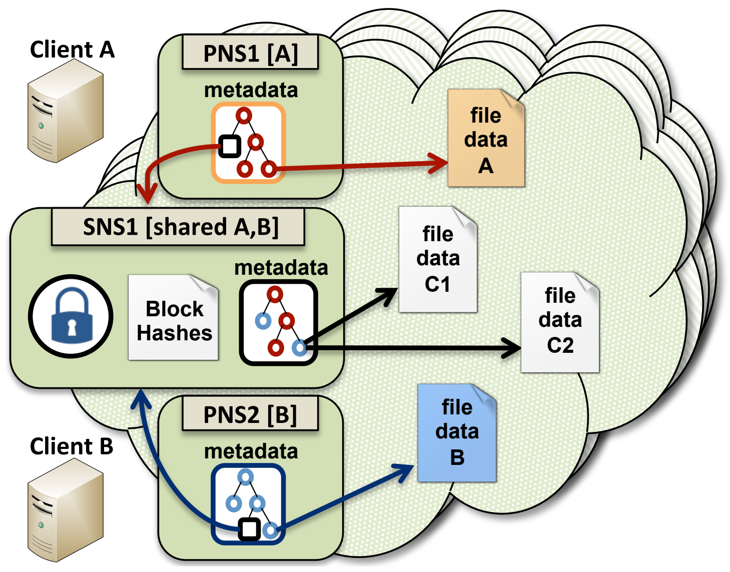


Figure A.2: Objects maintained in the cloud by CHARON.

only be accessed by their owner. Each client has only one PNS associated with it. On the other hand, a client has access to as many SNSs as the shared folders it can access. Each shared folder is associated to exactly one SNS, which is referenced in the PNSs of the clients sharing it.

Figure A.2 illustrates how a set of files relate with these namespaces. Files A and B are private to their owners (clients A and B, respectively), while file C is divided in two data blocks and is shared among the two clients. Since files A and B are private, their metadata is kept in their owner PNSs. In the case of file C, the reference to the file blocks is stored in SNS1.

Although similar, personal and shared namespaces differ in the way the hashes of the most recent versions of the files' data blocks are stored. In PNSs this information is kept together with the remaining files' metadata. When the PNS needs to be stored, these hashes are serialized together with the other metadata of the files and then uploaded to the clouds. On the other hand, in a SNS these hashes are placed in a separate *Block Hashes* (BH) object (see Figure A.2), as discussed bellow.

Another important difference between a PNS and a SNS is that the latter is associated with a lease, which must be acquired before any update is executed on a file or directory in the namespace. Notice that reading a SNS does not require locking, as the DepSky algorithm supports multiple readers even with concurrent writes.

Dealing with shared files. The PNS' metadata is downloaded from the cloud-of-clouds only once, when the file system is mounted. SNSs, on the other hand, are periodically fetched to find metadata updates on shared directories. The reason for having a separate BH (*Block Hashes*) object in SNSs comes from this need. Since a SNS contains all its files' metadata, having the hashes of all blocks of all files together with this information could significantly increase the monetary and performance costs of periodically downloading it. Given this, the BH object is only refreshed when a file is opened (either for read or write), to check if the particular file was updated. Moreover, storing the hashes of all data blocks in a single BH object can be costly due to fact that the size of such object could grow linearly with the number of file blocks stored in the SNS. To circumvent this problem, CHARON defines a maximum number of entries allowed in each BH object (e.g., 100 hashes). When this number is reached, newer block' hashes are saved in additional objects. However, there is an exception to this rule: hashes of data blocks from the same file are always kept in a single BH. This is done to avoid fetching several BH objects when opening a large file.

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 the upload completes, the corresponding metadata (i.e., the SNS hosting the file) is updated in the cloud-of-clouds, and then the lease is released. Such background propagation is crucial for the usability of the file system, since it can take a significant time to complete. However, other clients may still perform read-only operations on directories and files belonging to the leased SNS, and are able to read the data associated with the latest uploaded metadata version.

A.5.2 Data Management

CHARON uses several techniques for efficient dealing with big data without requiring custom services on the cloud. This sections describes some of these techniques.

Multi-level cache. CHARON uses the local disk to cache the most recent files used by clients. Moreover, it also keeps a fixed small main-memory cache to improve subsequent data accesses over open files. Both the main memory and disk caches implement LRU (Least Recently Used) policies. The use of a cache not only improves performance, but also decreases the operational cost of the system. This happens because cloud providers charge data downloads, but usually uploads are free (as an incentive to send data to their facilities [3, 8, 14, 18]), which means that the monetary cost of operating CHARON corresponds roughly to the cost of the used storage space plus the traffic required to download new versions of files.

Working with data blocks. Managing large files in cloud-backed file systems brings two main challenges. First, reading (resp. writing) whole files from the cloud (as done in SCFS [31]) is impractical due to the high downloading (resp. uploading) latency. Second, big files might not fit in the (memory) cache employed in cloud-backed file systems for ensuring usable performance [31, 67, 15, 40].

CHARON addresses these challenges by splitting (large) files into fixed-size blocks. The block size is configurable but we currently use 16MB as this size offers an interesting balance between latency and throughput [30, 31]. This strategy is similar to what is done in other cloud-backed file systems, such as BlueSky [67] that uses 4MB data blocks. Furthermore, blocks are compressed (using a lightweight algorithm) before being uploaded to their respective storage locations.

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 sequential 4kB-block reads. 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. 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. In common scenarios (e.g., a single client updates its files from a single host), private files are expected to be maintained in the local disk cache to approximate the performance of a local file system.

Prefetching. The prominence of sequential reads in bioinformatics workflows (see §A.6.2) motivate the use of block prefetching. Our implementation uses a thread pool responsible for prefetching data blocks from any location as soon as a sequential read is identified. Currently, CHARON starts prefetching when half of a block is sequentially read. If in the meanwhile the file being prefetched is closed, all enqueued requests for that file are removed from the prefetching queue. Besides reading data in advance, an additional advantage of this technique is to have several TCP connections getting data in parallel, accelerating the download of the whole file.

Supporting diverse storage locations. The cloud-of-clouds location is the default option for CHARON, but users may select any of the other two location types to store their files. Locations are defined using a semantic cue [64], in which the pathname specifies the location of a particular object. Specifically, in CHARON, users are only able to use cues over folders at the time of its creation. For example, the path `/share/.Location=S3/data` informs that the directory `data` is in Amazon S3. Henceforth, each file (or folder) created in a folder with a location different from the default one, will be stored in that defined location.

The use of a *cloud-of-clouds* to store data brings benefits in terms of resilience to failures and avoids vendor lock-in problems [20]. CHARON resorted to DepSky [30] to ensure that stored files have their availability, integrity and confidentiality preserved even if a fraction of providers fail. *Availability* is achieved by storing the data in multiple clouds avoiding the provider dependency of several previous cloud-backed storage systems [15, 62, 67]. It uses data-centric Byzantine quorum protocols [55] that require a set of $3f + 1$ cloud services, with at most f of them suffering Byzantine failures. DepSky uses storage-optimal erasure codes to store only portions of a file on each cloud, making the storage of a file approximately 50% more costly than storing it on a single cloud (for $f = 1$). *Integrity* is provided by cross-verifying the validity of the stored blocks using hashes stored in all clouds. *Confidentiality* 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 [50].

For not-so-critical files, the provider-maintained redundancy and security of a *single public cloud* may be enough. Therefore, this option can be used to reduce the storage costs of such files.

The *private repository* is used to store files containing privacy-sensitive information that are subject to regulatory legislation. CHARON uses secure protocols (e.g., TLS) to transfer these files directly between private repositories when they are shared, without ever storing it in the cloud.

Independently on the location, when a file block is updated, we first create an object with the new version and then update the corresponding PNS or SNS/BH to reference it. Old versions are removed in background, by a garbage collector triggered periodically on each client (e.g., once a day) to delete non-referenced blocks the client owns.

A.5.3 Security Model

CHARON implements a security model where the owner of the file 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 object. The cloud-of-clouds 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 (recall that data is encrypted using secret sharing [50]).

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 its account. Moreover, sharing is allowed only at the granularity of directory subtrees (as in Dropbox). To give permission to others to access a directory, 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 using the provider API. Namely, to share a directory, the system creates a SNS and gives permission to the sharing users, updating also the owner's PNS.

Each CHARON user id has to be mapped to the corresponding cloud accounts. This mapping is kept together with the client PNS in the cloud-of-clouds. Since there is no centralized server informing clients about the arrival of other clients to the system, the discovery of new clients and shared directories has to be done by external means (e.g., main invitation, as in Dropbox).

A.6 Evaluation

We conducted three sets of experiments to evaluate CHARON and compare it with other storage systems. First, we discuss the latency 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 present the results of a novel benchmark based on representative bioinformatics workflows.

Experimental Environment. Our experimental environment is comprised by four Dell Power Edge R410 machines connected through a gigabit network. 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 and Java 1.7.0_67 (64-bits).

The three storage locations for CHARON were configured as follows. The cloud-of-clouds storage is composed by Amazon S3 (US), Windows Azure Storage (UK), Rackspace Cloud Files (UK), and Google Cloud Storage (US). For the single cloud storage, we use Amazon S3 (US), since it is the most widely used storage cloud. The private repository was either located in the disk of the client machine or in a different machine in the same LAN, depending on the experiment. For the composite lease, we employ additional cloud services such as Azure Queue [10], RackSpace Queue [9], Amazon DynamoDB [1], and Google Datastore [6]. Therefore, all cloud-of-clouds configurations consider $f = 1$.

We compare all CHARON storage locations with the ext4 local file system, the Linux' NFSv4 deployed in our cluster, and other cloud-backed file systems with the code available such as SCFS [31] and S3QL [15]. Similarly to CHARON, these two systems were configured to update the cloud in background, and in the case of SCFS, the coordination service replicas are deployed in four medium instances on Amazon EC2 (UK).

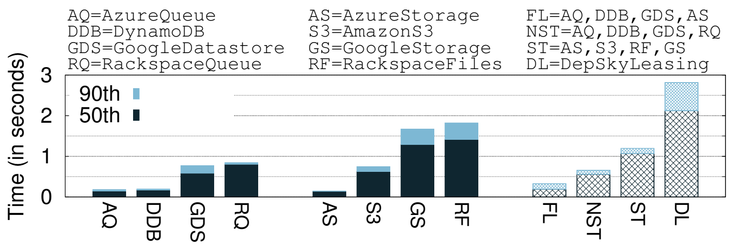


Figure A.3: Latency of lease (without contention) for several configurations of the composite and all base leases.

A.6.1 Composite Leasing

Our first set of experiments evaluate the composite lease algorithm described in §A.4.2 and the base objects used in its implementation. We focus our analysis on the lease operation, since it is the only one in the critical path of any application updating shared folders.

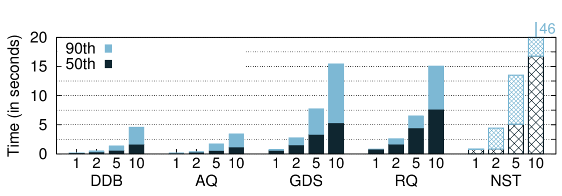
Contention-free executions. The composite leasing algorithm was configured in three ways: using only storage cloud services from different providers (ST), using only non-storage cloud services, such as queues (NST), and using the fastest base services, even if from the same providers (FL). Additionally, we compare these compositions with DepSky mutual exclusion algorithm (DL) [30], using the same services as the ST configuration. Figure A.3 presents the lease latency of these algorithms and their base lease objects.

The base lease objects require between 200 ms to 1.8 s to acquire a lease. Overall, the results for non-storage base lease objects are better than the ones using storage services. We hypothesize this happens because storage services are throughput-oriented, and thus less efficient when dealing with small objects such as lease entries.

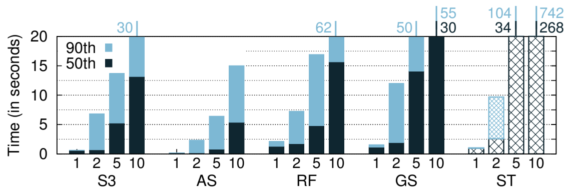
The results for the composite lease configurations reflect the performance of their base objects. More specifically, the lease protocol waits for a quorum of $2f + 1 = 3$ leasing acknowledgements from different services, which means that the latency of the composite lease is similar to the third fastest 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 FL configuration uses the fastest base objects available (storage and non-storage), achieving a latency 100% lower than in 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. Consequently, we use the NST configuration in CHARON 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.

The observed latency for DepSky locking (DL) is twice the latency of ST and four times bigger than NST (used in CHARON). This happens because the DepSky algorithm accesses the storage clouds in phases, and not by executing base lease algorithms in parallel.

Executions under contention. An 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 backoff time between 0-1 second. The results for several base lease objects and composite lease configurations are presented in Figure A.4.



(a) Non-storage services.



(b) Storage services.

Figure A.4: Latency of lease acquisition under contention of up to 10 clients for diverse algorithms and cloud services.

Again, non-storage services (Figure A.4(a)) provide better/faster results than storage services (Figure A.4(b)). This happens because most non-storage services satisfy the deadlock-freedom property, i.e., if several processes tries to get the lease concurrently, some process will succeed [25]. This makes them much better when dealing with contention than the storage-based algorithms, which only implement obstruction-freedom (see Table A.1).

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

A.6.2 File System Microbenchmarks

In this section we present the results of a set of microbenchmarks using Filebench [5] for comparing CHARON with other file systems.

Metadata-intensive operations. Our first experiment focus on how well the system deals with metadata intensive operations when compared with other systems. Table A.2 presents the number of operations per second for ext4 (on SSD), NFS, S3QL [15], SCFS [31], and CHARON. We use 0-byte files to focus on metadata management.

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

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

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

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

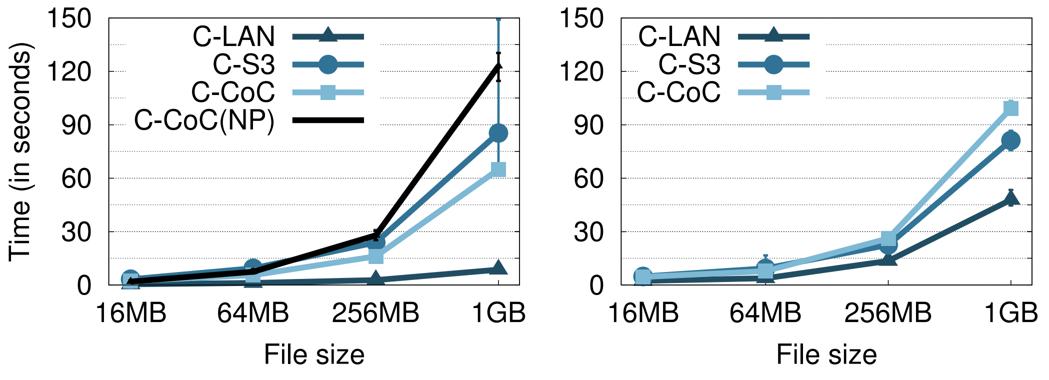
Operation	ext4	NFS	S3QL	SCFS	CHARON
seqRead	215	211	214	200	194
randRead	210	205	207	191	186
seqWrite	125	125	10	17	36

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

Not surprisingly, ext4 offers the best read-throughput both for sequential and random workloads. S3QL and NFS provide a lower, but similar, read throughput (sequential and random) when compared with ext4. SCFS and CHARON, despite presenting a lower performance, are still competitive for read workloads.

When considering write throughput, ext4 and NFS present the best performance for sequential workloads. However, CHARON presents at least $2\times$ better sequential write throughput than the other cloud-backed file systems. In particular, S3QL provides a write-throughput almost $4\times$ lower than our system. This happens because S3QL does not perform well when writing small chunks [16] (we use 8kB-writes).

Read and write of big files. Efficiency in reading and writing large files to/from the clouds is one of the main objectives of CHARON. Figure A.5 shows the time required for sequentially read and write non-cached files (from 16MB to 1GB). We performed these experiments considering different data locations for CHARON, namely: private repository in the same network (C-LAN), single cloud in Amazon S3 (C-S3), and the cloud-of-clouds (C-CoC). In the last scenario, the read operation is performed with and without prefetching (C-CoC(NP)). Differently from previous experiments, here the write latency includes the time required to upload the data to its final location.



(a) Non-cached read.

(b) Write and upload.

Figure A.5: Non-cached read (download) and write (upload) latencies for different file sizes and CHARON 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 for both reading and writing results. For writing, the additional latency presented by the cloud-of-clouds comes from the fact that we need to update the data in a Byzantine quorum of clouds (three out of four) to finish the write, thus the end-to-end latency will be dictated by the third fastest cloud. Finally, the results showed that prefetching file blocks significantly improves sequential reads of big files. As shown in the graph, downloading a 1GB-file from the clouds using prefetching decreases the whole-file read latency in 25%.

File sharing. The next microbenchmark compares the latency of sharing a file in CHARON and in other sharing-capable cloud-backed storage systems such Dropbox and SCFS. In this way, we repeated the sharing experiment introduced in [31], which consider desktop-application file sizes (256kB – 16MB). The experiment consists in a client writing a randomly-generated file in a shared folder while another client, located in the same LAN and with access to the folder, tries to read the written file. We measure the time it takes from the instant a client closes a newly written file until another client reads it entirely (and notifies the first client with an UDP message). Figure A.6 presents the results.

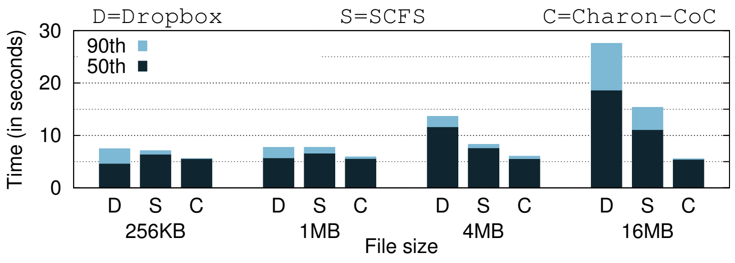


Figure A.6: Sharing latency of different cloud-backed systems for four desktop-size files.

The three evaluated systems provide a very similar latency with small files (256kB and 1MB). However, when the file size increases (4MB and 16MB), CHARON becomes the fastest system and Dropbox the slowest. For bigger files (not shown in the figure), this behavior becomes even more evident as Dropbox transfer data very slowly and SCFS assume files as single blocks. This happens because CHARON uses compression and erasure codes, reducing significantly the amount of transmitted data, and consequently the experienced latency.

A.6.3 Bioinformatics Workflows

In this section we present results considering a novel benchmark (dubbed FS-Biobench, fully described in a companion report [24]) that emulates the I/O of eight common bioinformatics workloads, summarized in Table A.4.

Figure A.7 presents the execution time of each of the mentioned FS-BioBench workflows for ext4 on SSD (ext4), for NFS with a server and a client in the same LAN, for S3QL and for CHARON using a repository in different locations: SSD in the same machine (C-Local), hard disk in a server in the same LAN (C-LAN), Amazon S3 (C-S3), and cloud-of-clouds (C-CoC). SCFS results are not presented because it does not support the big files (up to 1GB) used in this benchmark. We executed every workflow ten times on each scenario and report average values. CHARON's and S3QL's caches were cleaned after each workflow execution, as all results would be similar to C-Local if files were cached.

Ext4 and NFS serves as basis of comparison in this experiment and, as expected, are usually faster in running the workflows. The time needed for CHARON to finish workflows `W1` and

Workflow	Input Files	Int. + Output Files	Description
W1.Genotyping	–	0+1 (24MB)	Write a single genotyping file.
W2.Sequencing	–	0+1 (1GB)	Write a single sequencing file in FASTQ format.
W3.Prospection	2 (1MB)	0+1 (4kB)	Prospect appropriate samples for a study from two MIABIS XML files
W4.Alignment	1 (1GB)	0+1 (960MB)	Search DNA reads from W2 in a reference, and write the alignment resu
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 V
W8.Methylation	1 (1GB)	2+1 (999MB+4kB)	Align DNA reads, and write a list of methylated positions.

Table A.4: Characteristics of the eight FS-BioBench workflows. All reads and writes are sequential. Output files are divided in two groups – intermediate and final results. File sizes were scaled-down for faster benchmark execution.

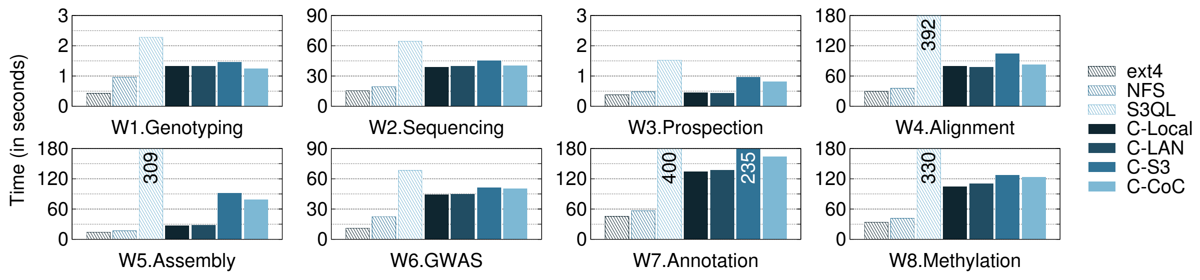


Figure A.7: FS-BioBench execution for different configurations of CHARON.

W2 is similar, independently from the data locations. This due to write operations immediately return after the file is updated in the local disk. Workflow W3 requires almost $2\times$ more time to finish in C-S3 and C-CoC than in C-Local and C-LAN due to the latency of fetching the two small files from the remote cloud services.

Workflows W4, W5, W7, and W8 are the ones requiring more time to run since they 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-Local is the fastest, followed by the C-LAN, the C-CoC, and the C-S3. Interestingly, running the benchmark in a cloud-of-clouds-hosted repository brings improvements in the running time of the benchmark (compared to C-S3) due to the capability of fetching blocks from the two fastest clouds at the moment [30]. Importantly, even considering that the latency for fetching input files dominate most of these benchmarks, in a real setup, the file processing can start as soon as the first block is available.

In conclusion, CHARON runs the workflows up to 250% (W4) faster than the other evaluated cloud-backed file system (S3QL). Furthermore, our system with the cloud-of-clouds is at most 150% slower than NFS in all workflows but W5 (which is read-only). This is an excellent result if one considers that the latency of accessing the cloud is $100\times$ higher than accessing the LAN-based NFS server.

A.7 Related Work

Bioinformatics and biobanks data sharing. There are several research infrastructures for maintaining bioinformatics datasets and making them available to researchers. For example, BBMRI [60] integrates several European biobanks [72] through a distributed research infrastructure. CGHub [70] is an American research center that stores cancer-related data that can be used by approved users. In all these infrastructures, there is one or more facilities responsible for managing a web-accessible repository of the data. CHARON aims to make research infrastructures like those (usually paid with public money) less expensive, more manageable, more dependable and widely-available by exploiting public clouds, without creating any dependency on these services. Furthermore, by working at the file system level, we make it easier to integrate any type of data with existing tools, without worrying about the specificities from each bioinformatics workflow.

Distributed file systems. CHARON borrows many ideas from existent file systems, such as the separation of data and metadata from NASD [42], volume leases from AFS [46], and background updates from several peer-to-peer file systems [21, 51, 64]. 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 [21]. Another related system is the xFS [22], a serverless network file system in which the storage of all data and metadata is done at the client side, with accesses coordinated through distributed protocols.

A fundamental difference between these systems and CHARON is that in our system clients do not communicate directly for coordination, but interact using widely-available untrusted cloud services. Furthermore, our design focuses 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 by our primary audience (e.g., non-system specialists). This decision differentiates our work from systems like Coda [48] and OriFS [58], which implements application-based conflict resolution. Finally, these systems do not explore the scalability and competitive prices of cloud storage, using instead the storage available on clients and servers.

Data-centric coordination. A centerpiece of CHARON's design is the use of Byzantine-resilient data-centric algorithms for implementing storage and coordination. There are some works that propose the use of this kind of algorithms for implementing dependable systems [19, 30, 34, 54]. At high level, our composite leasing is very similar to the at-most-one mutual exclusion of Phalanx, which resembles classical quorum-based mutual exclusion algorithms from the 80s. However, this algorithm requires servers to run custom code [54], while ours is built on top of non-replicated algorithms especially designed for the cloud services currently available.

Byzantine disk Paxos [19] is a Byzantine consensus protocol based 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, this solution would require a large number of cloud accesses, i.e., at least five per round, and multiple rounds might be needed in case of contention. Our composite protocol, on the other hand, requires between two to four sequential cloud access for acquiring a lease (see Table A.1).

To the best of our knowledge, there are only two fault-tolerant data-centric lease algorithms proposed in the literature [30, 34]. Both of them are based on the use of storage clouds or disks. The lease algorithm of [34] has two important differences when compared with CHARON's Byzantine-resilient composite lease. First, 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, which is adequate for implementing leader election but not locks. Second, it tolerates only crashes, requiring thus some trust on individual cloud providers. The BFT mutual exclusion algorithm presented in [30] is a natural candidate to regulate access contention in CHARON. However, our composite lease algorithm is up to four times faster than DepSky's (as shown in §A.6.1) and does not require clients to have synchronized clocks.

Cloud-backed storage solutions. There are many commercial storage proxies to transparently integrate local systems with cloud storage (e.g. [11, 12]). Most of them follow an architecture similar to BlueSky [67] and Iris [62], where a CIFS/NFS proxy is used to mediate access to a single cloud provider. In general, such systems do not support the coordination of file accesses between different proxies, and thus are inappropriate to share geographically-dispersed data.

In the last years, many works have been proposing the use of multiple cloud providers for improving the integrity and availability of stored data [20, 29, 30, 49, 71, 31, 40]. The idea was first introduced in archival systems like SafeStore [49] and RACS [20], with the latter requiring no modifications to the storage clouds or servers running in the cloud. More recent systems like ICStore [29] and DepSky [30] provide object storage (i.e., variable-size read/write registers) considering different fault models and unmodified storage clouds.

SPANStore [71] is a system that tries to optimize data placement and consistency guarantees taking into account several metrics of cloud providers (e.g., cost and access latency). The main limitation of this approach is that it requires servers deployed in all cloud providers, which implies additional costs and management complexity. The same limitation applies to modern (single-provider) geo-replicated storage systems such as Spanner [36] and Pileus [65], if deployed in multiple cloud providers.

Some recent cloud-backed storage systems employ a hybrid approach in which unmodified cloud storage services are used together with few computing nodes for storing metadata and coordinating data access [31, 40]. SCFS, in particular, provides controlled sharing of desktop-size files stored in multiple providers by using a cloud-deployed coordination service to implement locking [31].

When compared with these works, CHARON is unique in its ability to provide controlled sharing of big data without relying on any cloud-deployed server or centralized proxy.

A.8 Conclusions

We presented CHARON, a cloud-backed file system for big data sharing and storage that targets the needs for Biobanks and other life sciences institutions. The design of CHARON relies on two important principles: files metadata and data are stored in multiple cloud providers, without requiring trust on any of them individually, and the system is completely serverless. This latter principle lead us to design a novel Byzantine-resilient leasing protocol to avoid write-write conflicts that requires no custom server on the cloud. Our results show that such kind of 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.

Currently, the system is being integrated in an open-source PaaS for storage, processing and sharing of bioinformatics data. This platform will start to be deployed (experimentally) later this year. For the future, we want to investigate how deduplication can be integrated in our system using techniques like convergent dispersal [52].

Bibliography

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

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

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

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

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

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

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

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

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

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

[11] Nasuni UniFS. http://www.nasuni.com/.

[12] Panzura CloudFS. http://panzura.com/.

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

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

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

[16] S3QL 1.13.2 documentation: Known issues. http://www.rath.org/s3ql-docs/issues.html.

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

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

[19] 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.

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

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

[22] 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.

[23] Anonymized. Byzantine-resilient composite leasing – formalization and correctness proofs. http://sites.google.com/site/charoncompanion/, 2015.

[24] Anonymized. FS-BioBench: A file system benchmark from bioinformatics workflows. http://sites.google.com/site/charoncompanion/, 2015.

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

[26] T.K. Attwood, A. Gisel, N-E. Eriksson, and E. Bongcam-Rudloff. Concepts, historical milestones and the central place of bioinformatics in modern biology: A European perspective. In Dr. Mahmood A. Mahdavi, editor, *Bioinformatics - Trends and Methodologies*. 2011.

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

[28] 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.

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

[30] 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.

[31] 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.

[32] 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.

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

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

[35] Cloud Harmony. Service status. `https://cloudharmony.com/status-1year-of-storage-group-by-regions-and-provider`.

[36] James Corbet et. al. Spanner: Google's globally distributed database. *ACM Transactions on Computer Systems*, 31(3):8:1–8:22, August 2013.

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

[38] 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.

[39] EU Directive. 95/46/EC of the European Parliament and of the Council of 24 October 1995 on the protection of individuals with regard to the processing of personal data and on the free movement of such data. *Official Journal of the EC*, 23(6), 1995.

[40] D. Dobre, P. Viotti, and M. Vukolic. Hybris: Robust hybrid cloud storage. *SoCC*, 2014.

[41] George Gaskell, Martin W Bauer, et al. *Genomics and Society:" Legal, Ethical and Social Dimensions"*. Routledge, 2013.

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

[43] 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.

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

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

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

[47] 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.

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

[49] Ramakrishna Kotla, Lorenzo Alvisi, and Mike Dahlin. SafeStore: A durable and practical storage system. In *USENIX ATC*, 2007.

[50] Hugo Krawczyk. Secret sharing made short. In *Proc. of the 13th Int. Cryptology Conference – CRYPTO'93*, pages 136–146, August 1993.

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

[52] Mingqiang Li, Chuan Qin, Patrick P. C. Lee, and Jin Li. Convergent Dispersal: Toward Storage-Efficient Security in a Cloud-of-Clouds. In *Proc. of the 6th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 14)*, June 2014.

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

[54] 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.

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

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

[57] Benedikt Martens, Marc Walterbusch, and Frank Teuteberg. Costing of cloud computing services: A total cost of ownership approach. In *System Science (HICSS), 2012 45th Hawaii International Conference on*, pages 1563–1572. IEEE, 2012.

[58] 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.

[59] 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.

[60] 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.

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

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

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

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

[65] Douglas B. Terry, Vijayan Prabhakaran, Ramakrishna Kotla, Mahesh Balakrishnan, Marcos K. Aguilera, and Hussam Abu-Libdeh. Consistency-based service level agreements for cloud storage. In *Proc. of the 24th ACM Symposium on Operating Systems Principles – SOSP'13*, pages 309–324, 2013.

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

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

[68] 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.

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

[70] 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.

[71] 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 24th ACM Symposium on Operating Systems Principles – SOSP'13*, pages 292–308, 2013.

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